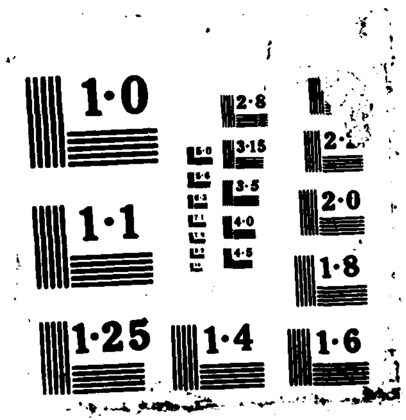
AD-A193 199 BASIC DATABASE OPERATIONS ON THE BUTTERFLY PARALLEL PROCESSOR: EXPERIMENT RESULTS(U) NAVAL RESEARCH LAB WASHINGTON DC T J ROSENAU ET AL. 04 MAR 88 NRL-MR-6173 1/1

UNCLASSIFIED F/G 12/7 NL


1·0
2·8
3·15
2·2
3·5
2·0
1·1
4·0
4·5
1·8
1·25
1·4
1·6

NRL Memorandum Report 6173

Basic Database Operations on the Butterfly Parallel Processor: Experiment Results

TODD J. ROSENAU AND SUSHIL JAJODIA

Computer Science and Systems Branch
Information Technology Division

March 4, 1988

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NRL Memorandum Report 6173	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Research Laboratory 6b. OFFICE SYMBOL (If applicable) Code 5576	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Washington, DC 20375-5000	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Office of Naval Research 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 63223C PROJECT NO. TASK NO WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
Basic Database Operations on the Butterfly Parallel Processor; Experiment Results

12. PERSONAL AUTHOR(S)
Rosenau, Todd J., and Jajodia, Sushil

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim	FROM 1/87 TO 12/87	1988 March 4	60

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Parallel processing; Butterfly computer; Relational databases; Hashing

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The next phase in speeding up database queries will be through the use of highly parallel computers. This paper will discuss the basic database operations (select, project, natural join, and scalar aggregrates) on a shared-memory multiple instruction stream, multiple data stream (MIMD) computer and the problems associated with implementing them. Some problems associated with getting maximum parallelization are improper data division and "hot spots." Improper data division results when the number of tasks does not divide evenly among the processors. Hot spots or contentions occur due to locking if accesses are made to the same segment of a RAMFile and also if attempts are made to get data from the same remote processor at the same time. These algorithms have been implemented on the Butterfly Parallel Processor, and the results of our experiments are described in detail.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Todd J. Rosenau	22b. TELEPHONE (Include Area Code) (202) 767-3107 22c. OFFICE SYMBOL Code 5576

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603

CONTENTS

BASIC DATABASE OPERATIONS ON THE BUTTERFLY PARALLEL PROCESSOR: EXPERIMENT RESULTS

1. Introduction

With the development of highly parallel machines such as the Butterfly parallel processor [3,4,5,9] and the Connection Machine [8,10], the time has come for the database researchers to look at ways to utilize these machines. Since some databases are extremely large, the time to process a query even with a large uniprocessor mainframe can reach unreasonable limits. Parallel machines offer substantial opportunities for performing these queries much faster than is possible with just a single processor. A uniprocessor must access the disk sequentially which can consume a significant amount of time. With the falling cost of main memory, uniprocessors and parallel processors are being created with very large amounts of RAM, some on the order of 16 to 512 Mbytes, with a Gigabyte not far off into the future. A multi-processor that allocates individual memory segments with each processor has the potential to store complete files in main memory by scattering them across the distinct memories of all the processors. The Butterfly computer allows you to load complete files into main memory by using the RAMFile system. File size should not be a problem since the available main memory can approach 1 Gigabyte (with 256 processors each with 4 Megabytes of memory).

Ideally, by utilizing ρ concurrent processors to perform a task one should be able to speed up its performance by a factor of ρ. However, there are many difficulties—hardware related as well as algorithmic in nature—which inhibit this improvement in computation speed. The usefulness of a parallel machine depends on the amount of parallelism inherent in a problem and how well this parallelism fits the architecture of the parallel machine [2,10]. Some serial algorithms do not have a suitable parallel analog since they contain some inherent timing constraints by depending on the results of previous calculations. Moreover, some parallel machine architectures are better suited for implementing certain parallel algorithms than others [10]. Hardware related issues include processor synchronization, memory access contention, and the design of the parallel machine (i.e., the number of processors, the way they are interconnected, whether or not they share a common memory, and so on).

The use of computers with large amounts of main memory will have a beneficial effect on database operations by allowing several files to be kept resident in memory while different queries progress.

Manuscript approved December 15, 1988.

Instead of having to keep swapping in and out different portions of a file from a external disk you can access the file immediately from RAM. This will save on disk accesses and if there is enough main memory you can even simulate a disk drive on each processor. Files will be stored in memory on the Butterfly computer by using the RAMFile system [5]. This method will not only store the file in main memory but will distribute the file across the different processors memory.

In this paper, we discuss the implementation of the basic database operations (select, project, natural join, and scalar aggregates) on a Butterfly parallel processor which is a shared-memory multiple instruction stream, multiple data stream (MIMD) computer. Some problems associated with getting maximum parallelization are improper data division and "hot spots." Improper data division results when the number of tasks does not divide evenly among the processors. Hot spots or contentions occur due to locking if accesses are made to the same segment of a RAMFile and also if attempts are made to to get data from the same remote processor at the same time. The results of our experiments are described in detail.

2. The RAMFile System

The RAMFile system is a utility that allows files to be loaded into main memory and accessed as a UNIX* file. The size of the file is constrained by the amount of main memory available, which can approach 1 Gigabyte, depending on the configuration. This is an upper limit because the operating system and other application programs will have to be resident in the same memory also. To load a file into and out of the Butterfly's main memory the application program must call a routine that will transfer the file from or to the host computer over the network using the streams-server. This can be slow serial process depending on the size of the file being loaded. This method appears to be a temporary solution until the butterfly can handle disk drives attached directly to it. The file will be distributed across the memory of all available processors in a round robin fashion, allocating blocks of the file to the next available processor with enough memory to hold that size block. It will continue "dealing" equal size segments of the file until the complete file has been dispursed. Each segment of a

* UNIX is a trademark of Bell Laboratories.

file can range in size from 256 bytes to 64 Kbytes in increments of powers of 2.

Access methods are very similar to UNIX system calls except that the prefix "RF_" is inserted in front of all system calls (for example, RF_create, RF_open, RF_seek, RF_read, RF_write, RF_close and so on). These calls are used exactly like their UNIX counterparts except for possible parameter differences. These primitives allow individual processes to access the RAMFile without worrying about contention and other related problems. The ability to address specific locations within a file comes from the use of the RF_seek command (similar to the lseek command). Upon opening a file via RF_open, all processes that must access the file can do so with all locking and mapping being hidden from the user.

The RAMFile system causes contention by locking a file on a segment basis. This means that if two processors try to access two different (or the same) memory locations that happen to lie within the same segment, only one processor will be able to proceed while the other will have to wait until the first unlocks the particular segment in the segment lock table. However, as we shall see, it is possible to reduce, sometimes even eliminate, such contention by choosing judiciously the size of the individual segments. A similar hardware related contention problem is that if two processors are trying to get data from the same remote processor through the Butterfly switch at the same time, one will have to wait for the first data transmission to complete.

BBN has included a systematic method for parallel programming, called the Uniform System Approach [4]. This approach has several methods for calling new processes and passing data between them. It simplifies the amount of work that the programmer has to do by allowing the programmer not to have to worry about actual task generation, segment allocation registers and other miscellaneous overhead. Its use has greatly simplified the development of these algorithms.

3. Hardware Description

The Butterfly Parallel Processor [3] is a MIMD machine with a shared memory. It can be configured with 1 to 256 processors, each of which is a Motorola MC68000. Each node contains 1 processor capable of 500,000 instructions per second. So a Butterfly with 256 Processors can compute at

128 MIPS. The main memory at each node is expandable from 1 to 4 Megabytes, so the maximum total memory on a 256 processor Butterfly is 1 Gigabyte. All nodes can be considered identical except for possible additional floating point processors which will enhance the performance of those processors and the existence of a king node which will contain the network software and other system maintenance software.

The configuration that was available to us during this experiment was a 32 processor Butterfly with 4 Mbytes of memory at each node, and all processors have been upgraded to Motorola MC68020 chips. This should double the processor speed to 1 MIPS per processor, giving a combined speed of 32 MIPS on our configuration. In order to measure the performance of our parallel algorithms, we chose to use a database similar to the one described in the Wisconsin Benchmark [6]. The database consisted of three relations containing 100, 1,000 and 10,000 tuples respectively. All attributes will have a predefined range of values that can occur in a specific field, and that range will have an even distribution of occurrences. For example, the hundred field will contain only the values 0 - 99 with an even distribution of values throughout the relation. If there are 1,000 tuples in a relation, then for each specific value of the hundred field there will be exactly 10 occurrences throughout the relation. Each relation will contain at least two unique integer fields that act as keys, depending on the field and the number of tuples in a relation there may be more. Also there will be three character string fields each with a length of 50 characters. The size of each tuple was 190 bytes long.

4. The Projection Operation

The projection operation will be broken down into three parts: file division, projection of attributes and the removal of duplicates. The file will be first scattered across the memory of the Butterfly and be accessed through the use of the RAMFile system [5] that is provided by BBN. Then the actual process of projection will take place followed by the removal of all duplicates.

4.1. File Division

When a file is loaded or created on the Butterfly computer, it is distributed across all the shared memories in equal sized segments. The file can be manipulated without knowledge of any RAMFile

parameters (segment size, file size, number of segments or what processor a particular segment lies); however, by using some a priori knowledge of the record sizes and other related processing decisions, it will be possible to adjust the RAMFile distribution parameters which will optimize access and execution times. This will be explained below.

Assuming the file can be thought of as a continuous block of memory with concurrent access, the first question is how to divide the file up among the tasks. Some definitions before continuing: let ρ be the number of processors, λ be the number of tasks to be operated on by the ρ processors, τ be the number of tuples in a relation, θ be the tuple size in bytes and δ be the number of tuples to be worked on by each task. Initially the process of projection will be done by breaking the file to be operated on down into ρ sections which will be worked on by ρ concurrent processes, so that $\lambda = \rho$. Since the file has been distributed among the ρ processors, and every tuples is accessible to every processor, a good initial value for δ would be:

$$\delta = \lceil \tau/\lambda \rceil = \lceil \tau/\rho \rceil$$

This will allow every processor to work on at most $\lceil \tau/\lambda \rceil$ of the tuples. The ceiling function is necessary for the case when τ is not a multiple of λ. Without the ceiling function, the remainder of τ/λ would go into task number $\lambda + 1$, causing $\rho - 1$ processors to wait while 1 processor finishes task number $\lambda + 1$. This can be very wasteful as can be seen in Figs. 1a and 1b, without and with the ceiling function. These figures were generated by first using an event logger that is available on the Butterfly and then viewing the events using the "gist" program.

One of the tools available on the Butterfly is an event logger. The event logger is useful for time stamping when certain events occur within the execution of an application. Before certain events can be logged they must be defined and a catalog must be declared to store the events on each processor. The overhead involved with logging events is quite minimal so they should not alter the performance of an application. This capability is extremely useful in detecting where and when different processes halt and where they are spending too much time. After the execution of the program and all events have been logged, the event log file must be down-loaded to the host computer. The next step is to view the events using the "gist" program. It will display the events on a two dimensional

Figure 1a: 1,000 tuple relation, 24 processors, without the ceiling function.

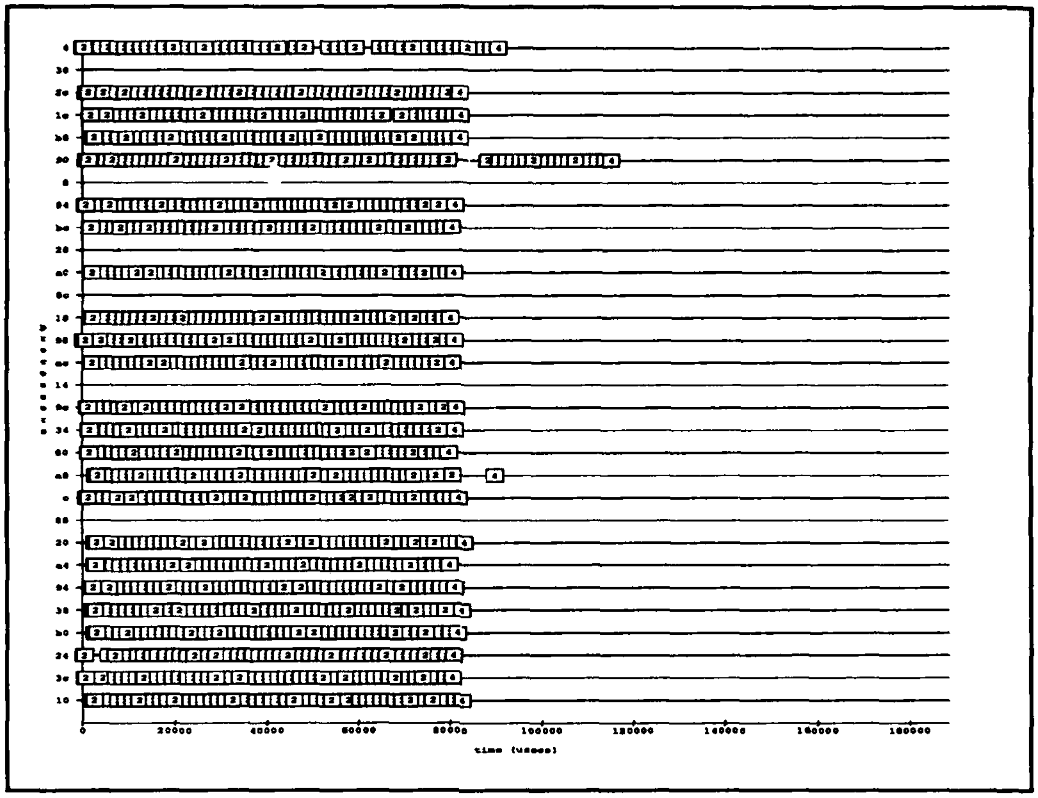
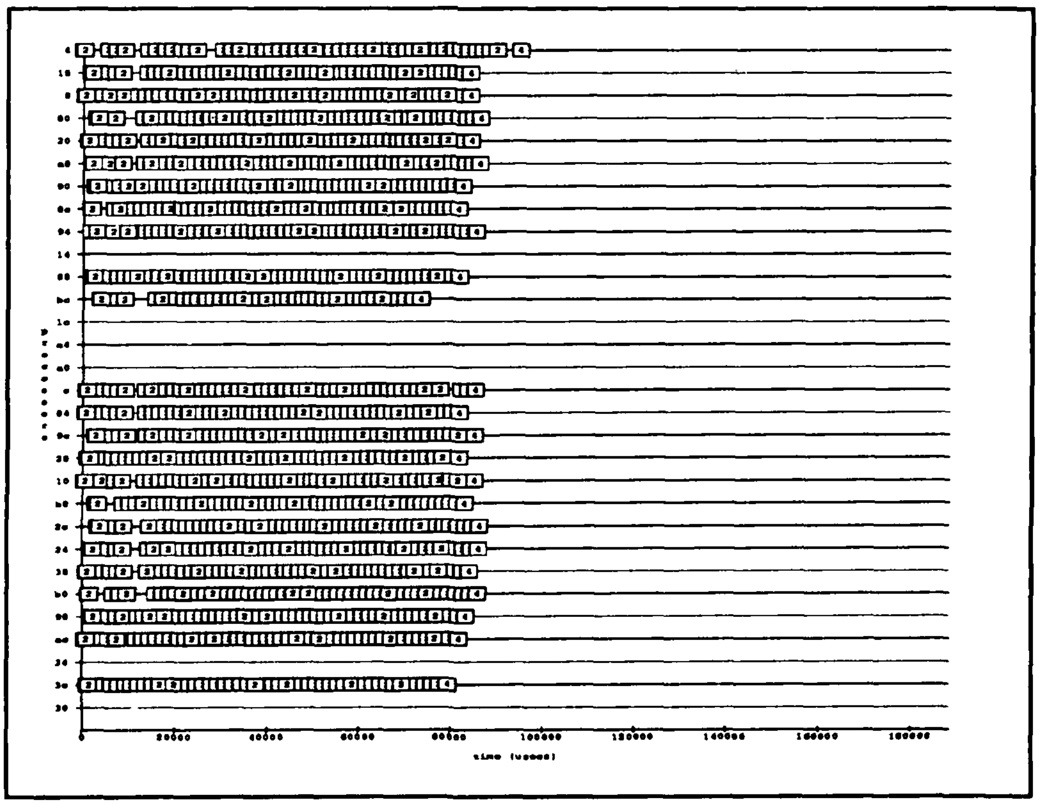


Figure 1b: 1,000 tuple relation, 24 processors, with the ceiling function.



graph with time on the horizontal axis and each processor's event history on the vertical axis. Without the ceiling function all processors must wait for one processor to finish its extra task in Fig. 1a, whereas with the ceiling function they all finish at about the same time (Fig. 1b). Some of the attractions of the ceiling function are that it is simple to compute and efficient in that there are no unnecessary task startups. One problem associated with this simple technique is that all tasks are *not* guaranteed to start at precisely the same time and therefore may *not* finish at the same time. Even those that do start at the same time are not guaranteed to finish together. This makes some nodes sit around idle waiting for the other nodes to finish their computations. If the size of a task becomes too long, there can be a considerable gap between when the first task finishes and when the last one finishes. There should be ways to keep all processors as active as possible without keeping any waiting around for others to finish. One way of keeping the waiting period to a minimum is to keep the size of the tasks to a "reasonable" length. One problem here is that if the size of the tasks gets to be too short then the overhead necessary to run and start each task can become greater than the actual run time of the task.

4.2. Projection of Attributes

The actual projection of this file will be done in λ separate tasks. Each task will operate on its own block of the file, with size δ. It will simply loop through its own block of the file, reading a tuple and then writing out only the attributes that are to be projected. The placement of the projected tuple within the output RAMFile will have a substantial effect on the performance of this algorithm.

In the current implementation of the Butterfly RAMFile system, a file must have a predeclared static length upon creation. This means that the output file must be large enough to hold the output of a query in the worst case. For a projection this is fine because there will be τ tuples in the initial output file before removal of duplicates. Now, a simple way to write tuples to the output file would be to just create a file and then keep appending them to the end of the file. This can be simulated with a predeclared length file by creating a EOF pointer that will point to the record that is currently at the end of the file. It must be a shared variable because all processors must be able to access it, and also it must be able to be atomically incremented. For something to be atomically operated on means

to have complete control of it without having to worry about concurrency problems. It turns out that this is a poor idea since it creates contention for both the EOF counter variable as well the current last segment in the output file. The latter contention occurs when a segment is large enough to hold more than one tuple. As an example, if a segment could hold 4.5 tuples, then the 5 or 6 processors that have tuples that must be placed in that particular segment must serially write to that segment. The 6 processors comes from the fact that the 4.5 tuples could be laid out in this manner: 0.25, 1, 1, 1, 1, 0.25. Since the output file must be predeclared, these contentions can be avoided by placing each output tuple in the same position as in the input file. Tuple t_i where i is the position of the tuple in the file, $0 \leq i < \tau$ will be placed in the ith position in the output file. By removing the contention for the EOF allows tuple placement to proceed much faster as can be seen in Figs. 2a, 2b and 2c.

4.3. Projection Results

By varying several parameters and the number of processors working on a task, the Butterfly Parallel Processor was able to give impressive speedups in two ways. One direction of speedup was going from one processor to several (our configuration went up to 30 processors available to the user). The other direction resulted from fine tuning the program's parameters. What will follow here will be a progression of parametric tuning that will increase the efficiency of the application. The initial parameters are: input file segment sizes are 256 bytes for the 1,000 and 100 tuple files and 512 bytes for the 10,000 tuple file, output file segment size is 256 bytes, the ceiling function will NOT be used and the EOF pointer WILL be incorporated. Duplicates have not been been removed, this will be covered later.

4.3.1. EOF Pointer Variable

As discussed in section 3.2, ideally it would seem logical to just keep appending tuples to the end of the output file. This would allow all processes to place all tuples into just one file without any worry about where to insert the next one. On a single processor computer there is a file header that contains information about the file such as: size, location and the current position within the file. As tuples are appended the current position pointer is increased by the size of the write or read. On the

Figure 2a: 10,000 Tuples

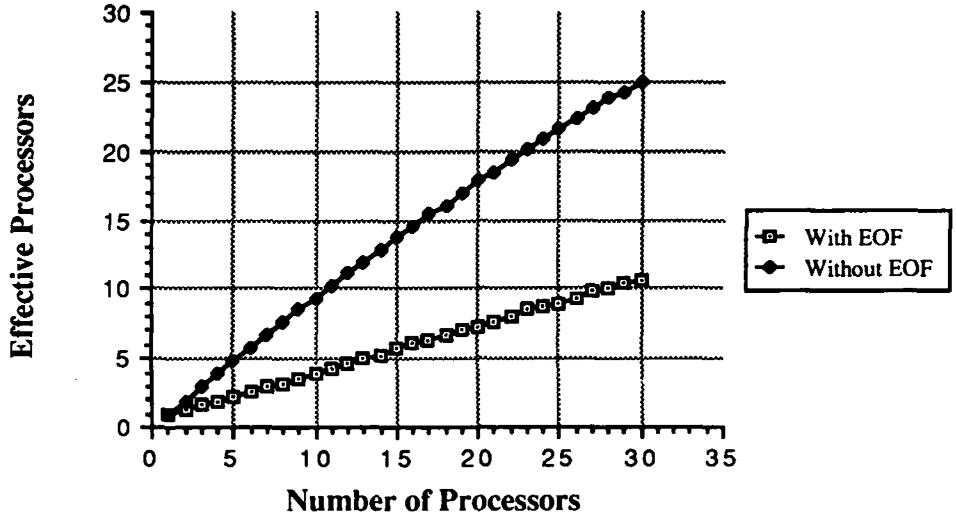


Figure 2b: 1,000 Tuples

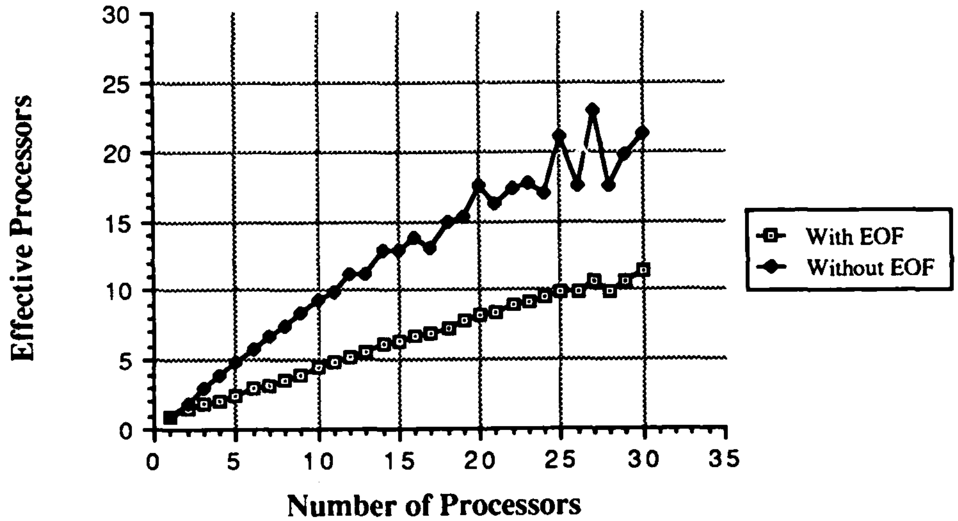
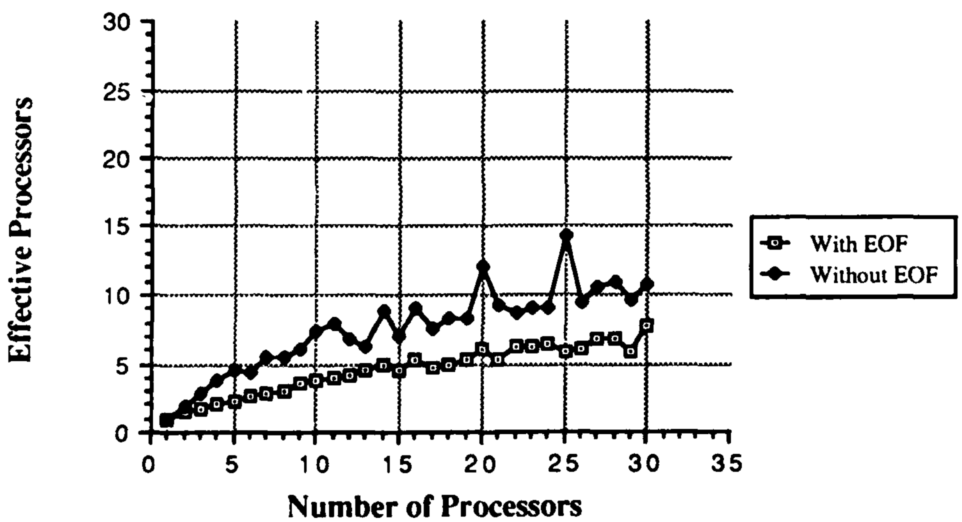


Figure 2c: 100 Tuples



Butterfly each processor has its own copy of the file descriptor which includes it own current position pointer. This creates three different problems, two of which are contention related and the third is an extra unnecessary computation.

Table 1 10,000 Tuple Projection with EOF Pointer			
No. of Processors	Time	Effective Processors	Efficiency
1	29.28	1.0	1.0000
2	21.15	1.3	0.6921
3	18.16	1.6	0.5372
4	15.23	1.9	0.4805
5	12.89	2.2	0.4541
6	11.24	2.6	0.4341
7	9.99	2.9	0.4186
8	8.90	3.2	0.4112
9	8.02	3.6	0.4053
10	7.33	3.9	0.3991
11	6.72	4.3	0.3957
12	6.21	4.7	0.3928
13	5.77	5.0	0.3897
14	5.43	5.3	0.3850
15	5.02	5.8	0.3884
16	4.75	6.1	0.3845
17	4.56	6.4	0.3776
18	4.29	6.8	0.3784
19	4.14	7.0	0.3719
20	4.00	7.3	0.3658
21	3.76	7.7	0.3702
22	3.61	8.0	0.3681
23	3.44	8.5	0.3700
24	3.29	8.8	0.3697
25	3.22	9.0	0.3635
26	3.09	9.4	0.3642
27	2.98	9.8	0.3631
28	2.90	10.0	0.3603
29	2.81	10.4	0.3587
30	2.71	10.7	0.3593

Table 2 10,000 Tuple Projection without EOF Pointer			
No. of Processors	Time	Effective Processors	Efficiency
1	27.54	1.0	1.0000
2	13.86	1.9	0.9930
3	9.29	2.9	0.9876
4	7.01	3.9	0.9818
5	5.65	4.8	0.9740
6	4.74	5.8	0.9679
7	4.08	6.7	0.9634
8	3.59	7.6	0.9572
9	3.21	8.5	0.9529
10	2.91	9.4	0.9453
11	2.66	10.3	0.9380
12	2.45	11.2	0.9364
13	2.27	12.0	0.9299
14	2.12	12.9	0.9248
15	2.00	13.7	0.9180
16	1.89	14.5	0.9104
17	1.77	15.4	0.9102
18	1.70	16.1	0.8960
19	1.61	17.0	0.8955
20	1.54	17.8	0.8925
21	1.48	18.5	0.8817
22	1.42	19.3	0.8798
23	1.36	20.1	0.8757
24	1.31	20.8	0.8706
25	1.26	21.7	0.8681
26	1.22	22.4	0.8630
27	1.18	23.1	0.8574
28	1.15	23.8	0.8500
29	1.13	24.3	0.8396
30	1.10	24.9	0.8332

The two contention problems are very related with one being overshadowed by the other. All RAMFiles have in their master file descriptor a segment lock table. Since RAMFiles are spread across all the processors' memory in equal sized segments, there must be a method to maintain file consistency. File accesses are kept consistent by locking the file on a segment basis. A read or write that is local to just one segment will first lock that segment, read/write and then unlock the segment. If the read or write operation will overlap segments then it will lock, read/write and then unlock, seg-

Table 3 1,000 Tuple Projection with EOF Pointer			
No. of Processors	Time	Effective Processors	Efficiency
1	3.59	1.0	1.0000
2	2.30	1.5	0.7791
3	1.96	1.8	0.6115
4	1.69	2.1	0.5319
5	1.42	2.5	0.5057
6	1.23	2.9	0.4873
7	1.09	3.2	0.4687
8	0.98	3.6	0.4578
9	0.87	4.0	0.4549
10	0.80	4.4	0.4488
11	0.73	4.8	0.4427
12	0.67	5.3	0.4450
13	0.64	5.5	0.4297
14	0.57	6.2	0.4447
15	0.56	6.3	0.4217
16	0.52	6.8	0.4284
17	0.51	6.9	0.4084
18	0.49	7.2	0.4034
19	0.45	7.9	0.4189
20	0.43	8.2	0.4118
21	0.43	8.3	0.3980
22	0.40	8.9	0.4066
23	0.38	9.2	0.4014
24	0.37	9.5	0.3969
25	0.36	9.9	0.3967
26	0.36	9.9	0.3825
27	0.33	10.7	0.3975
28	0.36	9.8	0.3516
29	0.33	10.7	0.3702
30	0.31	11.4	0.3830

Table 4 1,000 Tuple Projection without EOF Pointer			
No. of Processors	Time	Effective Processors	Efficiency
1	3.43	1.0	1.0000
2	1.74	1.9	0.9852
3	1.16	2.9	0.9802
4	0.88	3.9	0.9766
5	0.70	4.8	0.9757
6	0.58	5.8	0.9777
7	0.51	6.7	0.9572
8	0.45	7.5	0.9391
9	0.40	8.4	0.9392
10	0.36	9.4	0.9435
11	0.34	9.8	0.8958
12	0.30	11.2	0.9397
13	0.30	11.2	0.8654
14	0.26	12.8	0.9199
15	0.26	12.8	0.8578
16	0.24	13.8	0.8686
17	0.26	13.0	0.7698
18	0.23	14.9	0.8304
19	0.22	15.2	0.8017
20	0.19	17.5	0.8761
21	0.21	16.2	0.7740
22	0.19	17.4	0.7914
23	0.19	17.7	0.7717
24	0.20	16.9	0.7067
25	0.16	21.1	0.8472
26	0.19	17.5	0.6735
27	0.14	23.0	0.8534
28	0.19	17.5	0.6278
29	0.17	19.8	0.6847
30	0.16	21.3	0.7109

ment by segment. Using just one current position pointer for all processors to maintain the current EOF will cause all write requests to be localized around the EOF pointer. This will cause a backlog of requests for the EOF pointer, the segment lock table and the actual transfer of data to and from the file. The first two operations must be completed atomically and the actual transfer of data will be atomic on a segment by segment basis. All of these atomic requests for global memory will be overshadowed by the fact that all memory requests that are resident in one processor's memory must proceed atomically. This is independent of the fact that the locations requested are different. The Butterfly switch hardware dictates that at any particular processor, only one processor may access that processor's memory at a time.

Table 5 100 Tuple Projection with EOF Pointer			
No. of Processors	Time	Effective Processors	Efficiency
1	0.36	1.0	1.0000
2	0.23	1.5	0.7689
3	0.20	1.7	0.5875
4	0.17	2.0	0.5104
5	0.15	2.3	0.4734
6	0.13	2.7	0.4558
7	0.12	2.8	0.4084
8	0.11	3.0	0.3773
9	0.10	3.5	0.3951
10	0.09	3.8	0.3842
11	0.09	4.0	0.3642
12	0.08	4.1	0.3496
13	0.07	4.5	0.3509
14	0.07	5.0	0.3574
15	0.07	4.5	0.3044
16	0.06	5.3	0.3338
17	0.07	4.8	0.2859
18	0.07	5.0	0.2789
19	0.06	5.3	0.2803
20	0.05	6.0	0.3021
21	0.06	5.3	0.2564
22	0.05	6.3	0.2885
23	0.05	6.3	0.2775
24	0.05	6.4	0.2698
25	0.06	5.8	0.2337
26	0.05	6.1	0.2379
27	0.05	6.8	0.2534
28	0.05	6.7	0.2424
29	0.06	5.9	0.2062
30	0.04	7.7	0.2576

Table 6 100 Tuple Projection without EOF Pointer			
No. of Processors	Time	Effective Processors	Efficiency
1	0.34	1.0	1.0000
2	0.17	1.9	0.9599
3	0.12	2.8	0.9346
4	0.09	3.7	0.9320
5	0.07	4.5	0.9120
6	0.07	4.4	0.7392
7	0.06	5.5	0.7962
8	0.06	5.5	0.6925
9	0.05	6.0	0.6678
10	0.04	7.4	0.7446
11	0.04	8.0	0.7323
12	0.05	6.7	0.5662
13	0.05	6.2	0.4796
14	0.03	8.9	0.6419
15	0.04	6.9	0.4604
16	0.03	9.0	0.5662
17	0.04	7.6	0.4470
18	0.04	8.3	0.4644
19	0.04	8.3	0.4387
20	0.02	12.0	0.6010
21	0.03	9.3	0.4468
22	0.03	8.6	0.3943
23	0.03	9.1	0.3958
24	0.03	9.1	0.3806
25	0.02	14.3	0.5732
26	0.03	9.5	0.3690
27	0.03	10.6	0.3930
28	0.03	11.0	0.3957
29	0.03	9.6	0.3314
30	0.03	10.7	0.3592

The third problem is caused by the fact that every processor will have its own copy of the file descriptor with its own current position pointer. If all current position pointers are equal and then one process writes to the end of the file, all of the other current position pointers will not be pointing to the end of the file. This can be overcome by having a global EOF pointer. Here all processes must atomically read and increment the pointer and then reseek to the new EOF. It would be nice if all RAMFile reads or writes would be independent of any other read or write by not trying to access the same segment (or region) at the same time. This simulated end of file is unnecessary because of the fact that a blank file has already been created to a predefined size and each processor has its own current position pointer. By scattering the locations to be read or written throughout the RAMFile we should be able to minimize the contention for the same segment (or region). Since every task will

be assigned an equal sized continuous block of tuples (except possibly the last block which may be smaller but still continuous), a solution would be to treat all blocks as ordered and all tuples within a block as ordered also and then simply place each task's projected tuple into its respective position within its respective block. All tasks' output blocks would also be in the same order as their input blocks. Once a task starts it will compute the index of the first tuple in its block to be projected. It can do this since it will know what block number it is assigned, the size of each block and the total file size. Then it will seek to this tuple's position in the input file and to its respective position in the output file. Instead of placing each tuple at the end of file, it will place the projected tuple into its respective position. Since each processors' file position pointer is incremented after each read/write by the size of the buffer being read/written, only one seek address must be computed and positioned to, the first position in each block, compared to every new global EOF position each time there is a read or write performed by any processor. This will save on computations and unnecessary seek operations. If we look at the execution times (Tables 1 through 6) for doing projections of 10,000, 1,000 and 100 tuples with and without the global EOF pointer on 1 through 30 processors we will see that the global EOF pointer is very inefficient.

Assuming all tasks start at the same time and progress at the same rate, then they will be reading and writing to positions that are on the average δ tuples apart. If δ is large enough then by the time one task is reaching the end of its block (and approaching the beginning of the next) the task working on that approaching block should have progressed towards the end of its block also thus keeping a reasonable distance between task's read and write positions.

4.3.2. Input File Segment Size

The next step in making the projection operation on the Butterfly more efficient will involve variations of the RAMFile parameters. These changes will improve overall execution times. That is, the speed up in the performance is not through parallelization; the execution times will improve greatly for runs with only one processor also.

Since RAMFiles are distributed across all memories in predeclared equal sized segments, each segment will not be guaranteed to contain integral tuples. Depending on the size of the tuple and the

Table 7 10,000 Tuple Projection With Different Input File Segment Sizes						
Input File Segment Size:	512 Bytes		1,024 Bytes		2,048 Bytes	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	27.54	1.0	24.15	1.0	22.71	1.0
2	13.86	1.9	12.16	1.9	11.46	1.9
3	9.29	2.9	8.16	2.9	7.71	2.9
4	7.01	3.9	6.15	3.9	5.81	3.9
5	5.65	4.8	4.96	4.8	4.69	4.8
6	4.74	5.8	4.15	5.8	3.91	5.8
7	4.08	6.7	3.59	6.7	3.39	6.6
8	3.59	7.6	3.16	7.6	2.98	7.6
9	3.21	8.5	2.83	8.5	2.67	8.4
10	2.91	9.4	2.55	9.4	2.40	9.4
11	2.66	10.3	2.34	10.3	2.21	10.2
12	2.45	11.2	2.15	11.1	2.04	11.1
13	2.27	12.0	2.00	12.0	1.90	11.9
14	2.12	12.9	1.87	12.8	1.77	12.8
15	2.00	13.7	1.76	13.7	1.66	13.6
16	1.89	14.5	1.66	14.4	1.56	14.5
17	1.77	15.4	1.57	15.3	1.49	15.2
18	1.70	16.1	1.48	16.2	1.41	16.1
19	1.61	17.0	1.45	16.6	1.34	16.8
20	1.54	17.8	1.35	17.8	1.28	17.6
21	1.48	18.5	1.30	18.4	1.25	18.0
22	1.42	19.3	1.25	19.3	1.18	19.1
23	1.36	20.1	1.20	20.0	1.14	19.8
24	1.31	20.8	1.16	20.7	1.09	20.7
25	1.26	21.7	1.12	21.4	1.06	21.2
26	1.22	22.4	1.08	22.3	1.02	22.1
27	1.18	23.1	1.05	22.8	0.99	22.7
28	1.15	23.8	1.02	23.5	0.97	23.3
29	1.13	24.3	0.98	24.5	0.94	24.0
30	1.10	24.9	0.96	24.9	0.91	24.9

file segment size, tuples will overlap segment boundaries causing delays because a processor will have to lock a segment, read it, and then unlock it, once for each segment in which the tuple lies. The size of the tuples that are used in this paper are 190 bytes, and if the input file segment size is 256 bytes then there will be one or possibly two partial tuples residing in each segment. This is very inefficient because approximately two out of every three tuples will have to be read in two parts. It would be more efficient to increase the size of the segment so that it would hold more tuples. As the number of tuples that can reside in a segment increases, there will be fewer read operations that will have to be completed in several sections. Caution must be used in deciding on the right segment size, since the factors that must be taken into account are the number of tuples in a file, the tuple size and the

Table 8 10,000 Tuple Projection With Different Input File Segment Sizes						
Input File Segment Size:	4,096 Bytes		8,192 Bytes		16,384 Bytes	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	22.33	1.0	22.18	1.0	21.96	1.0
2	11.23	1.9	11.17	1.9	11.05	1.9
3	7.56	2.9	7.49	2.9	7.43	2.9
4	5.70	3.9	5.66	3.9	5.60	3.9
5	4.59	4.8	4.55	4.8	4.50	4.8
6	3.84	5.8	3.81	5.8	3.78	5.7
7	3.32	6.7	3.29	6.7	3.25	6.7
8	2.92	7.6	2.90	7.6	2.88	7.6
9	2.61	8.5	2.59	8.5	2.57	8.5
10	2.37	9.4	2.36	9.3	2.33	9.4
11	2.16	10.2	2.16	10.2	2.13	10.2
12	1.99	11.1	1.98	11.1	1.95	11.2
13	1.86	11.9	1.84	11.9	1.83	11.9
14	1.73	12.8	1.71	12.9	1.70	12.9
15	1.62	13.7	1.62	13.6	1.60	13.7
16	1.53	14.5	1.53	14.4	1.51	14.4
17	1.46	15.2	1.44	15.3	1.43	15.3
18	1.39	15.9	1.39	15.9	1.36	16.0
19	1.31	16.9	1.30	16.9	1.29	17.0
20	1.27	17.5	1.25	17.6	1.24	17.7
21	1.21	18.3	1.22	18.1	1.21	18.0
22	1.15	19.2	1.16	19.0	1.14	19.2
23	1.13	19.7	1.11	19.8	1.09	19.9
24	1.07	20.7	1.07	20.7	1.06	20.6
25	1.04	21.4	1.03	21.3	1.02	21.4
26	1.01	22.0	1.00	22.1	0.99	22.1
27	0.97	22.8	0.98	22.5	0.95	22.9
28	0.94	23.5	0.94	23.4	0.93	23.6
29	0.91	24.3	0.91	24.1	0.90	24.3
30	0.89	24.8	0.90	24.6	0.88	24.8

number of processors.

If the segment size gets too large then the number of segments will decrease causing a possible unbalanced distribution of the file between the different processors. This will happen when the number of segments approaches the number of processors. In this case some processors will receive a larger percentage of the file than others, causing possible contention problems. As this happens the segment size will approach the size of the block of tuples that each task will work on: *60*. When this happens two processors may start to operate on the same segment creating contention problems. As the input file segment size increases in Tables 7 through 10 notice that all of the times are decreasing but the

Table 9
1,000 Tuple Projection With Different Input File Segment Sizes

Input File Segment Size:	256 Bytes		512 Bytes		1,024 Bytes		2,048 Bytes	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	3.43	1.0	2.74	1.0	2.40	1.0	2.24	1.0
2	1.74	1.9	1.39	1.9	1.21	1.9	1.13	1.9
3	1.16	2.9	0.92	2.9	0.81	2.9	0.76	2.9
4	0.88	3.9	0.70	3.9	0.62	3.8	0.57	3.8
5	0.70	4.8	0.57	4.8	0.50	4.7	0.47	4.6
6	0.58	5.8	0.47	5.7	0.41	5.7	0.39	5.7
7	0.51	6.7	0.41	6.6	0.35	6.7	0.33	6.6
8	0.45	7.5	0.36	7.5	0.31	7.6	0.29	7.4
9	0.40	8.4	0.32	8.4	0.28	8.4	0.27	8.2
10	0.36	9.4	0.29	9.3	0.25	9.4	0.24	9.1
11	0.34	9.8	0.28	9.6	0.24	9.7	0.22	9.8
12	0.30	11.2	0.24	11.0	0.22	10.9	0.20	10.8
13	0.30	11.2	0.24	10.9	0.21	11.0	0.20	10.8
14	0.26	12.8	0.22	12.3	0.19	12.3	0.18	12.2
15	0.26	12.8	0.22	12.3	0.19	12.6	0.18	12.4
16	0.24	13.8	0.20	13.6	0.17	13.7	0.16	13.6
17	0.26	13.0	0.20	13.2	0.18	13.2	0.16	13.3
18	0.23	14.9	0.18	14.4	0.16	14.3	0.14	15.2
19	0.22	15.2	0.18	14.7	0.16	14.8	0.15	14.7
20	0.19	17.5	0.15	17.3	0.14	16.9	0.12	17.2
21	0.21	16.2	0.17	15.6	0.16	14.3	0.14	15.4
22	0.19	17.4	0.16	17.0	0.14	16.9	0.13	17.1
23	0.19	17.7	0.17	15.6	0.13	17.1	0.13	16.8
24	0.20	16.9	0.16	16.6	0.14	16.5	0.13	16.8
25	0.16	21.1	0.12	21.3	0.11	20.7	0.11	19.6
26	0.19	17.5	0.15	18.1	0.13	18.2	0.12	18.1
27	0.14	23.0	0.11	23.1	0.10	21.9	0.10	22.3
28	0.19	17.5	0.16	16.6	0.14	16.6	0.13	16.6
29	0.17	19.8	0.14	19.0	0.12	19.1	0.13	16.7
30	0.16	21.3	0.12	21.4	0.11	21.1	0.10	20.6

number of effective processors are staying about the same for runs with equal numbers of processors. By varying this parameter will make the algorithm run faster but not increase significantly the amount of concurrency being performed. An optimal segment size will be one that is approximately equal to no more than $\delta\theta/2$ in order to minimize the number of tuples residing on the border of two segments. To prevent contention, a segment should not contain any more than $\delta/2$ tuples. This is because as two adjacent tasks progress, there will be a buffer of at least one segment between them. Although $\delta\theta/2$ should be taken as an upper limit of the segment size, the number of tuples stored on each processor should not differ by any more than approximately fifty percent. We have chosen fifty percent since we want all data to be distributed across all processors as evenly as possible.

Table 10 100 Tuple Projection With Different Input File Segment Sizes								
Input File Segment Size:	256 Bytes		512 Bytes		1,024 Bytes		2,048 Bytes	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	0.34	1.0	0.27	1.0	0.24	1.0	0.22	1.0
2	0.17	1.9	0.16	1.7	0.12	1.9	0.11	1.9
3	0.12	2.8	0.09	2.8	0.08	2.8	0.08	2.7
4	0.09	3.7	0.07	3.7	0.06	3.7	0.06	3.5
5	0.07	4.5	0.06	4.0	0.05	4.4	0.05	4.2
6	0.07	4.4	0.06	4.4	0.05	4.4	0.05	4.4
7	0.06	5.5	0.06	4.0	0.06	3.7	0.04	5.2
8	0.06	5.5	0.05	5.0	0.04	5.3	0.04	5.2
9	0.05	6.0	0.04	6.3	0.03	6.7	0.03	5.9
10	0.04	7.4	0.03	7.5	0.03	7.3	0.04	5.4
11	0.04	8.0	0.03	7.2	0.03	7.6	0.03	6.8
12	0.05	6.7	0.03	6.9	0.03	6.5	0.03	5.9
13	0.05	6.2	0.05	5.3	0.04	5.5	0.04	4.9
14	0.03	8.9	0.03	8.0	0.03	7.8	0.03	6.8
15	0.04	6.9	0.04	6.6	0.03	6.4	0.03	5.6
16	0.03	9.0	0.03	7.7	0.03	7.2	0.03	7.0
17	0.04	7.6	0.03	7.5	0.03	7.2	0.03	5.9
18	0.04	8.3	0.03	7.7	0.03	7.1	0.03	6.2
19	0.04	8.3	0.03	7.7	0.03	7.8	0.03	6.6
20	0.02	12.0	0.02	11.2	0.02	10.4	0.02	8.0
21	0.03	9.3	0.04	6.6	0.03	7.7	0.03	5.8
22	0.03	8.6	0.03	8.6	0.03	7.8	0.03	5.6
23	0.03	9.1	0.03	8.5	0.03	7.3	0.03	7.2
24	0.03	9.1	0.03	7.4	0.02	8.2	0.03	6.9
25	0.02	14.3	0.02	11.8	0.02	10.1	0.02	8.2
26	0.03	9.5	0.03	9.1	0.03	7.3	0.03	6.3
27	0.03	10.6	0.03	9.0	0.02	8.1	0.03	5.9
28	0.03	11.0	0.04	6.1	0.03	7.5	0.03	6.2
29	0.03	9.6	0.02	9.5	0.02	8.6	0.03	6.0
30	0.03	10.7	0.03	9.0	0.02	8.1	0.03	6.8

Tables 7 - 10 show that as the input file segment size approaches its optimal size, the execution times for the 10,000 and 1,000 tuple projections decrease. For the 100 tuple relation the smallest possible segment size (256 bytes) was optimal so as we vary the segment size from 256, 512, 1024 and 2048 bytes, the execution times for runs with a single processor decrease while runs with maximum processors increases or become more sporadic.

4.3.3. Output File Segment Size

The reasons for varying the size of the output file segment size are the same as for the input file except for the fact that the output file may be the input file for the next step in the query. An

example of this is that this file may be just an intermediate file to be used as the input to the removal of duplicates algorithm. If the number of processors working on different portions of a query will vary then what may be optimal for one set of processors may not be optimal for another. See tables 11 through 14. The input file segment sizes will be 16,384 bytes for 10,000 tuple relation, 2,048 bytes for 1,000 tuple relation and 256 bytes for 100 tuple relation.

Table 11
10,000 Tuple Projection With Different Output File Segment Sizes

Output File Segment Size:	256 Bytes		512 Bytes		1024 Bytes	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	21.96	1.0	18.80	1.0	17.15	1.0
2	11.05	1.9	9.47	1.9	8.63	1.9
3	7.43	2.9	6.35	2.9	5.80	2.9
4	5.60	3.9	4.80	3.9	4.38	3.9
5	4.50	4.8	3.87	4.8	3.53	4.8
6	3.78	5.7	3.24	5.7	2.97	5.7
7	3.25	6.7	2.80	6.7	2.56	6.6
8	2.88	7.6	2.47	7.6	2.26	7.5
9	2.57	8.5	2.21	8.5	2.03	8.4
10	2.33	9.4	2.00	9.3	1.83	9.3
11	2.13	10.2	1.83	10.2	1.68	10.1
12	1.95	11.2	1.69	11.1	1.55	11.0
13	1.83	11.9	1.56	11.9	1.44	11.8
14	1.70	12.9	1.47	12.7	1.34	12.7
15	1.60	13.7	1.38	13.6	1.26	13.5
16	1.51	14.4	1.30	14.3	1.20	14.2
17	1.43	15.3	1.23	15.1	1.13	15.0
18	1.36	16.0	1.17	16.0	1.07	15.8
19	1.29	17.0	1.12	16.6	1.03	16.5
20	1.24	17.7	1.08	17.3	0.98	17.3
21	1.21	18.0	1.02	18.3	0.94	18.2
22	1.14	19.2	0.98	18.9	0.90	18.9
23	1.09	19.9	0.95	19.6	0.87	19.5
24	1.06	20.6	0.91	20.5	0.84	20.2
25	1.02	21.4	0.88	21.3	0.81	21.0
26	0.99	22.1	0.86	21.7	0.78	21.8
27	0.95	22.9	0.84	22.3	0.77	22.2
28	0.93	23.6	0.80	23.3	0.74	23.0
29	0.90	24.3	0.78	23.9	0.71	23.8
30	0.88	24.8	0.77	24.3	0.70	24.4

4.3.4. Ceiling Function

The problems associated with the ceiling function were discussed in section 3.1. See tables 15 through 20. The size of the last extra task (without the ceiling function) will be equal to:

Table 12 10,000 Tuple Projection With Different Output File Segment Sizes						
Output File Segment Size:	2048 Bytes		4096 Bytes		8192 Bytes	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	16.29	1.0	15.97	1.0	15.80	1.0
2	8.21	1.9	8.06	1.9	8.00	1.9
3	5.51	2.9	5.41	2.9	5.36	2.9
4	4.19	3.8	4.09	3.9	4.04	3.9
5	3.37	4.8	3.30	4.8	3.26	4.8
6	2.83	5.7	2.78	5.7	2.73	5.7
7	2.44	6.6	2.39	6.6	2.37	6.6
8	2.15	7.5	2.11	7.5	2.09	7.5
9	1.93	8.4	1.89	8.4	1.87	8.4
10	1.76	9.2	1.71	9.3	1.69	9.3
11	1.60	10.1	1.56	10.1	1.55	10.1
12	1.47	11.0	1.45	10.9	1.43	11.0
13	1.38	11.7	1.34	11.8	1.33	11.8
14	1.28	12.6	1.26	12.6	1.25	12.6
15	1.21	13.4	1.18	13.4	1.17	13.4
16	1.15	14.0	1.12	14.2	1.10	14.2
17	1.08	14.9	1.06	14.9	1.05	15.0
18	1.03	15.7	1.01	15.7	1.00	15.7
19	0.99	16.4	0.96	16.5	0.95	16.5
20	0.94	17.2	0.92	17.2	0.91	17.3
21	0.90	18.0	0.88	18.0	0.87	18.0
22	0.87	18.6	0.85	18.6	0.84	18.6
23	0.83	19.4	0.81	19.5	0.81	19.4
24	0.80	20.2	0.79	20.1	0.78	20.1
25	0.78	20.8	0.76	20.7	0.77	20.4
26	0.75	21.4	0.74	21.3	0.74	21.3
27	0.73	22.2	0.72	22.0	0.71	22.0
28	0.71	22.6	0.69	22.8	0.69	22.8
29	0.69	23.4	0.69	23.0	0.67	23.3
30	0.67	24.1	0.66	23.9	0.65	24.0

$$\delta_{\rho+1} = \tau - \rho \left\lfloor \frac{\tau}{\rho} \right\rfloor$$

Since the maximum value of $\delta_{\rho+1}$ will be: $\delta_{\rho+1} < \rho$, it may be very small compared to the actual value of δ. As can be seen with the projections of 100 and 1,000 tuples, there is a noticeable increase in efficiency. Not so with the 10,000 tuple projection, the value of $\delta_{\rho+1}$ will be very small compared to the computed value of delta. With our current configuration ($1 \leq \rho \leq 30$) thus ($333 \leq \delta \leq 10{,}000$). We shall see later that there is another reason why the 10,000 tuple projection was prevented from showing the benefit from the ceiling function.

Table 13 1,000 Tuple Projection With Different Output File Segment Sizes						
Output File Segment Size:	256 Bytes		512 Bytes		1,024 Bytes	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	2.24	1.0	1.96	1.0	1.82	1.0
2	1.13	1.9	0.99	1.9	0.92	1.9
3	0.76	2.9	0.66	2.9	0.64	2.8
4	0.57	3.8	0.50	3.8	0.47	3.8
5	0.47	4.6	0.40	4.8	0.38	4.7
6	0.39	5.7	0.34	5.6	0.31	5.7
7	0.33	6.6	0.29	6.5	0.27	6.6
8	0.29	7.4	0.26	7.4	0.24	7.4
9	0.27	8.2	0.23	8.3	0.22	8.2
10	0.24	9.1	0.21	9.1	0.19	9.1
11	0.22	9.8	0.20	9.6	0.19	9.5
12	0.20	10.8	0.18	10.8	0.17	10.7
13	0.20	10.8	0.18	10.6	0.17	10.5
14	0.18	12.2	0.15	12.3	0.14	12.2
15	0.18	12.4	0.15	12.5	0.15	12.0
16	0.16	13.6	0.14	13.5	0.13	13.4
17	0.16	13.3	0.14	13.2	0.14	12.9
18	0.14	15.2	0.13	14.5	0.12	14.0
19	0.15	14.7	0.14	13.6	0.12	14.2
20	0.12	17.2	0.11	16.8	0.10	16.8
21	0.14	15.4	0.12	15.6	0.11	15.2
22	0.13	17.1	0.11	16.8	0.11	15.9
23	0.13	16.8	0.11	17.0	0.11	16.4
24	0.13	16.8	0.12	16.1	0.11	15.7
25	0.11	19.6	0.10	19.6	0.08	20.4
26	0.12	18.1	0.10	18.0	0.10	17.4
27	0.10	22.3	0.09	21.8	0.08	21.7
28	0.13	16.6	0.11	16.6	0.11	16.4
29	0.13	16.7	0.10	18.8	0.10	18.1
30	0.10	20.6	0.09	21.3	0.09	19.9

4.3.5. One Processor Running Slower

After making many improvements to the projection algorithm, the event logger was employed to see if there was any other places where improvements could be made. It was here that it was noticed that one processor (the parent processor) was running substantially slower than the rest of the processors. This problem was experienced when projecting the larger relation (10,000 tuples) and was only noticed occasionally with the 1,000 tuple relation. The event log for projecting a 10,000 tuples relation running on 30 processors is shown in Fig. 3.

Processor execution times may vary for several reasons. There may be other processes running

Figure 3: Speed ratio = 1.00

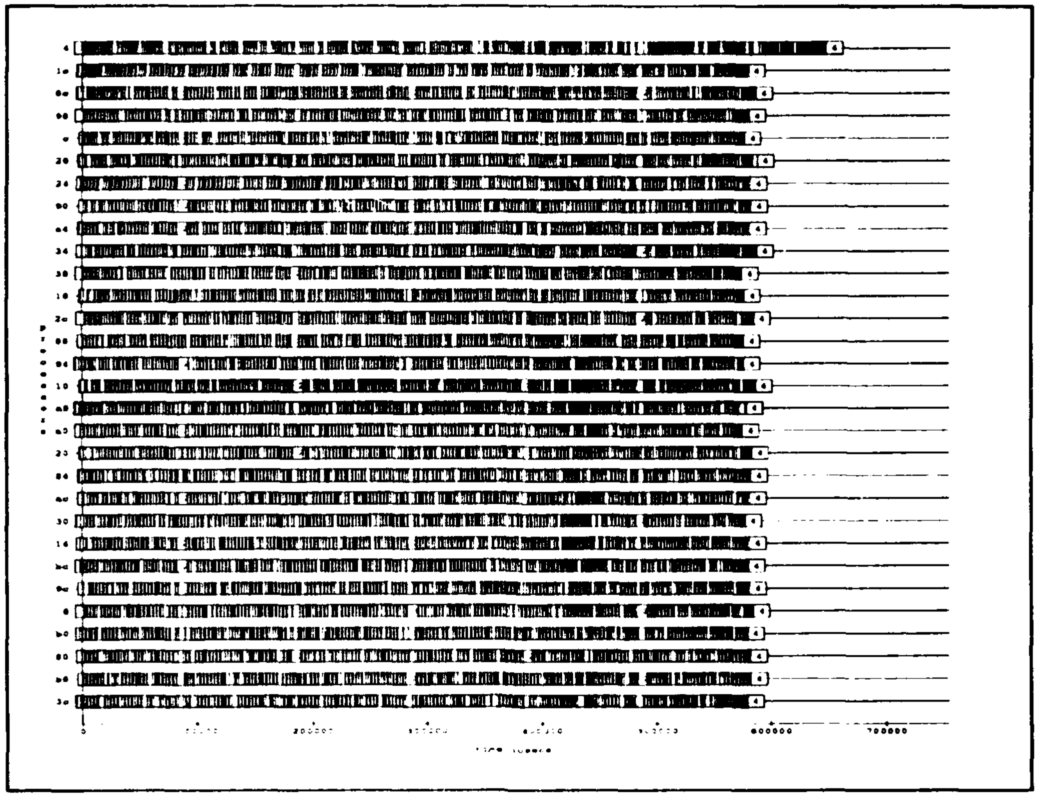


Figure 4: Speed ratio = 0.85

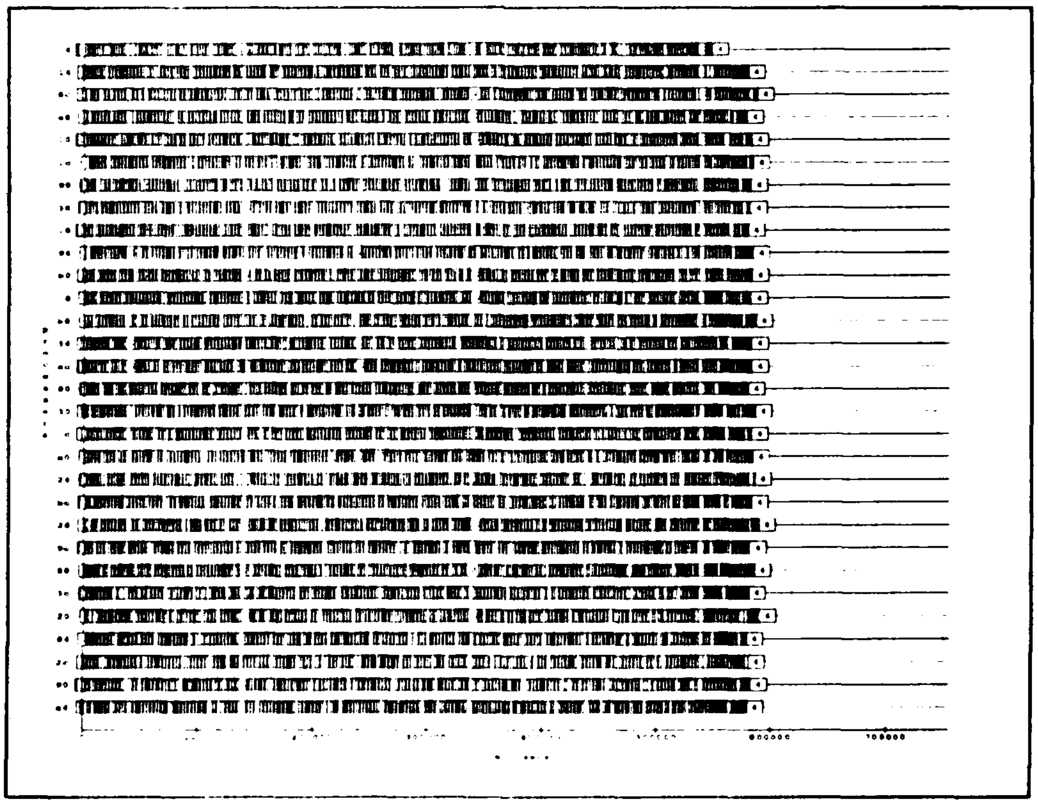


Table 14 100 Tuple Projection With Different Output File Segment Sizes						
Output File Segment Size:	256 Bytes		512 Bytes		1,024 Bytes	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	0.24	1.0	0.21	1.0	0.20	1.0
2	0.12	1.9	0.11	1.8	0.12	1.6
3	0.08	2.8	0.07	2.8	0.08	2.4
4	0.06	3.7	0.06	3.5	0.05	3.5
5	0.05	4.4	0.05	3.7	0.04	4.3
6	0.05	4.4	0.05	4.3	0.05	4.0
7	0.06	3.7	0.04	4.9	0.04	4.1
8	0.04	5.3	0.04	4.9	0.04	4.7
9	0.03	6.7	0.03	5.9	0.03	5.9
10	0.03	7.3	0.03	6.9	0.03	5.6
11	0.03	7.6	0.03	7.0	0.03	5.9
12	0.03	6.5	0.03	6.1	0.03	5.6
13	0.04	5.5	0.04	5.2	0.04	4.8
14	0.03	7.8	0.03	6.7	0.03	5.4
15	0.03	6.4	0.03	5.8	0.03	5.3
16	0.03	7.2	0.03	6.7	0.03	6.1
17	0.03	7.2	0.04	4.5	0.04	4.5
18	0.03	7.1	0.03	6.1	0.03	5.1
19	0.03	7.8	0.02	7.2	0.04	4.2
20	0.02	10.4	0.03	6.2	0.03	6.4
21	0.03	7.7	0.03	6.3	0.04	4.8
22	0.03	7.8	0.03	6.6	0.05	3.5
23	0.03	7.3	0.02	7.5	0.03	5.5
24	0.02	8.2	0.04	5.1	0.03	6.6
25	0.02	10.1	0.02	8.4	0.03	6.2
26	0.03	7.3	0.03	6.6	0.04	4.7
27	0.02	8.1	0.03	6.5	0.03	5.2
28	0.03	7.5	0.03	6.6	0.03	5.4
29	0.02	8.6	0.03	6.2	0.03	5.8
30	0.02	8.1	0.02	7.2	0.03	6.3

on a subset of the processors that are not a product of the current application, such as the window manager or other operating systems processes. Other reasons for a certain processor running at a slower speed could be contention. Contention can occur since only one processor can access a particular memory module at a time. If heavily used shared variables are all stored on one processor's memory, there will develop a backlog of access requests for those particular locations residing in one processor's memory.

With varying processor execution speeds it may be advantageous to give the faster processors slightly more tuples and consequently the slower processors slightly less tuples. Since only one proces-

Table 15 10,000 Tuple Projection Without Ceiling Function			
No. of Processors	Time	Effective Processors	Efficiency
1	15.80	1.0	1.0000
2	8.00	1.9	0.9871
3	5.36	2.9	0.9814
4	4.04	3.9	0.9766
5	3.26	4.8	0.9691
6	2.73	5.7	0.9629
7	2.37	6.6	0.9519
8	2.09	7.5	0.9452
9	1.87	8.4	0.9376
10	1.69	9.3	0.9313
11	1.55	10.1	0.9240
12	1.43	11.0	0.9202
13	1.33	11.8	0.9098
14	1.25	12.6	0.9016
15	1.17	13.4	0.8966
16	1.10	14.2	0.8904
17	1.05	15.0	0.8845
18	1.00	15.7	0.8757
19	0.95	16.5	0.8686
20	0.91	17.3	0.8657
21	0.87	18.0	0.8574
22	0.84	18.6	0.8495
23	0.81	19.4	0.8449
24	0.78	20.1	0.8409
25	0.77	20.4	0.8183
26	0.74	21.3	0.8214
27	0.71	22.0	0.8163
28	0.69	22.8	0.8163
29	0.67	23.3	0.8045
30	0.65	24.0	0.8012

Table 16 10,000 Tuple Projection With Ceiling Function			
No. of Processors	Time	Effective Processors	Efficiency
1	15.81	1.0	1.0000
2	7.99	1.9	0.9889
3	5.35	2.9	0.9839
4	4.05	3.9	0.9755
5	3.26	4.8	0.9687
6	2.74	5.7	0.9618
7	2.36	6.6	0.9543
8	2.09	7.5	0.9443
9	1.87	8.4	0.9349
10	1.69	9.3	0.9321
11	1.56	10.1	0.9203
12	1.43	11.0	0.9173
13	1.33	11.8	0.9084
14	1.25	12.5	0.8995
15	1.17	13.4	0.8976
16	1.11	14.2	0.8895
17	1.05	15.0	0.8841
18	1.00	15.7	0.8745
19	0.96	16.4	0.8664
20	0.92	17.1	0.8559
21	0.87	18.0	0.8591
22	0.84	18.6	0.8491
23	0.82	19.2	0.8368
24	0.78	20.0	0.8368
25	0.75	20.8	0.8352
26	0.73	21.4	0.8238
27	0.71	22.0	0.8185
28	0.69	22.8	0.8166
29	0.67	23.4	0.8078
30	0.66	23.8	0.7960

sor is running slower, we would like to distribute the slower processor's extra tuples that are remaining to be processed after all the other processors have finished their blocks, among the rest of the processors. Comparing the execution times for the faster and the slower processors will give us a ratio of their speeds, ϕ:

$$\phi = \frac{\textit{faster processors time}}{\textit{slower processors time}}$$

$$\phi \leq 1.0$$

This ratio of the processors' speed will give us the percentage $\phi\delta$ of δ, the number of tuples operated on by one processor, that will have been completed by the slower processor when the rest of the processors have completed their δ tuples. The task division should be arranged so that the slower

Table 17 1,000 Tuple Projection Without Ceiling Function			
No. of Processors	Time	Effective Processors	Efficiency
1	1.82	1.0	1.0000
2	0.92	1.9	0.9891
3	0.64	2.8	0.9463
4	0.47	3.8	0.9701
5	0.38	4.7	0.9571
6	0.31	5.7	0.9550
7	0.27	6.6	0.9428
8	0.24	7.4	0.9319
9	0.22	8.2	0.9204
10	0.19	9.1	0.9196
11	0.19	9.5	0.8657
12	0.17	10.7	0.8924
13	0.17	10.5	0.8142
14	0.14	12.2	0.8756
15	0.15	12.0	0.8057
16	0.13	13.4	0.8413
17	0.14	12.9	0.7644
18	0.12	14.0	0.7832
19	0.12	14.2	0.7521
20	0.10	16.8	0.8441
21	0.11	15.2	0.7275
22	0.11	15.9	0.7243
23	0.11	16.4	0.7170
24	0.11	15.7	0.6543
25	0.08	20.4	0.8175
26	0.10	17.4	0.6702
27	0.08	21.7	0.8047
28	0.11	16.4	0.5873
29	0.10	18.1	0.6271
30	0.09	19.9	0.6658

Table 18 1,000 Tuple Projection With Ceiling Function			
No. of Processors	Time	Effective Processors	Efficiency
1	1.84	1.0	1.0000
2	0.92	1.9	0.9951:
3	0.64	2.8	0.95241
4	0.48	3.7	0.9476
5	0.38	4.7	0.9544
6	0.32	5.6	0.9443
7	0.29	6.2	0.8929
8	0.24	7.5	0.9414
9	0.22	8.2	0.9134
10	0.20	9.0	0.9082
11	0.18	10.1	0.9248
12	0.17	10.6	0.8903
13	0.15	11.6	0.8946
14	0.14	12.5	0.8986
15	0.14	13.1	0.8768
16	0.13	13.3	0.8369
17	0.12	14.3	0.8435
18	0.12	14.9	0.8304
19	0.11	15.7	0.8291
20	0.11	15.9	0.7996
21	0.11	16.3	0.7801
22	0.10	17.7	0.8077
23	0.09	18.4	0.8041
24	0.09	19.3	0.8069
25	0.08	20.6	0.8249
26	0.08	20.6	0.7937
27	0.08	20.9	0.7774
28	0.08	22.3	0.7978
29	0.08	22.7	0.7833
30	0.07	23.6	0.7881

processor will only process at most $\phi\delta$ of the tuples. Depending on the integer value of δ, this will leave at least $(\tau - \phi\delta)$ for the $(\rho - 1)$ fast processors. These remaining tuples will be divided among the $(\rho - 1)$ processors as before. Thus the formula for computing the new block size with a speed ratio of ϕ is:

$$\delta = \left\lceil \frac{\tau - \left\lfloor \phi \left\lceil \frac{\tau}{\rho} \right\rceil \right\rfloor}{\rho - 1} \right\rceil$$

Which processor gets which block of tuples can be specified by first allocating the $(\rho - 1)$ full-sized blocks in parallel and then calling the last smaller sized block of tuples on the slower processor.

Table 19 100 Tuple Projection Without Ceiling Function			
No. of Processors	Time	Effective Processors	Efficiency
1	0.34	1.0	1.0000
2	0.17	1.9	0.9599
3	0.12	2.8	0.9346
4	0.09	3.7	0.9320
5	0.07	4.5	0.9120
6	0.07	4.4	0.7392
7	0.06	5.5	0.7962
8	0.06	5.5	0.6925
9	0.05	6.0	0.6678
10	0.04	7.4	0.7446
11	0.04	8.0	0.7323
12	0.05	6.7	0.5662
13	0.05	6.2	0.4796
14	0.03	8.9	0.6419
15	0.04	6.9	0.4604
16	0.03	9.0	0.5662
17	0.04	7.6	0.4470
18	0.04	8.3	0.4644
19	0.04	8.3	0.4387
20	0.02	12.0	0.6010
21	0.03	9.3	0.4468
22	0.03	8.6	0.3943
23	0.03	9.1	0.3958
24	0.03	9.1	0.3806
25	0.02	14.3	0.5732
26	0.03	9.5	0.3690
27	0.03	10.6	0.3930
28	0.03	11.0	0.3957
29	0.03	9.6	0.3314
30	0.03	10.7	0.3592

Table 20 100 Tuple Projection With Ceiling Function			
No. of Processors	Time	Effective Processors	Efficiency
1	0.34	1.0	1.0000
2	0.19	1.7	0.8958
3	0.12	2.8	0.9595
4	0.09	3.7	0.9461
5	0.07	4.5	0.9115
6	0.06	5.0	0.8429
7	0.05	5.8	0.8426
8	0.05	6.1	0.7684
9	0.04	6.9	0.7762
10	0.04	7.7	0.7727
11	0.04	7.5	0.6862
12	0.04	7.6	0.6359
13	0.03	9.1	0.7046
14	0.03	8.7	0.6283
15	0.03	9.3	0.6209
16	0.03	10.7	0.6688
17	0.04	7.7	0.4552
18	0.03	10.2	0.5702
19	0.03	10.5	0.5557
20	0.03	10.4	0.5239
21	0.02	13.0	0.6199
22	0.02	11.7	0.5362
23	0.02	12.6	0.5492
24	0.03	10.6	0.4442
25	0.02	12.9	0.5195
26	0.02	14.5	0.5600
27	0.02	14.4	0.5336
28	0.02	12.7	0.4563
29	0.02	14.0	0.4854
30	0.02	14.3	0.4790

The speed ratio of our slow processor to an average faster one in a 30 processor run was approximately 0.85 during a 10,000 tuple projection. Thus the slower processor will only have to project at most 285 tuples compared to 335 for the faster processors. Looking at the new event log (Fig. 4) for the next modification, you will see that all processors are finishing closer together minimizing idle time. Comparing the times (Tables 21 and 22) for the previous best run and the adaptation for a slower processor run we will see an drop of .10 seconds in execution time for a 30 processor run. This corresponds to an increase in the effective processors of 4.0. The ratio 0.85 was used for all runs with 2 through 30 processors thus accounting for a slight speed drop in the 2 through 4 processor runs. This can be fixed by computing a distinct speed ratio for runs with different processor counts.

Table 21
10,000 Tuple Projection
With Initial Block Size

No. of Processors	Time	Effective Processors	Efficiency
1	15.81	1.0	1.0000
2	7.99	1.9	0.9889
3	5.35	2.9	0.9839
4	4.05	3.9	0.9755
5	3.26	4.8	0.9687
6	2.74	5.7	0.9618
7	2.36	6.6	0.9543
8	2.09	7.5	0.9443
9	1.87	8.4	0.9349
10	1.69	9.3	0.9321
11	1.56	10.1	0.9203
12	1.43	11.0	0.9173
13	1.33	11.8	0.9084
14	1.25	12.5	0.8995
15	1.17	13.4	0.8976
16	1.11	14.2	0.8895
17	1.05	15.0	0.8841
18	1.00	15.7	0.8745
19	0.96	16.4	0.8664
20	0.92	17.1	0.8559
21	0.87	18.0	0.8591
22	0.84	18.6	0.8491
23	0.82	19.2	0.8368
24	0.78	20.0	0.8368
25	0.75	20.8	0.8352
26	0.73	21.4	0.8238
27	0.71	22.0	0.8185
28	0.69	22.8	0.8166
29	0.67	23.4	0.8078
30	0.66	23.8	0.7960

Table 22
10,000 Tuple Projection
Modified Block Size, Speed Ratio: 0.85

No. of Processors	Time	Effective Processors	Efficiency
1	15.84	1.0	1.0000
2	8.88	1.7	0.8913
3	5.56	2.8	0.9486
4	4.08	3.8	0.9701
5	3.24	4.8	0.9773
6	2.69	5.8	0.9812
7	2.30	6.8	0.9823
8	2.01	7.8	0.9827
9	1.78	8.8	0.9837
10	1.61	9.8	0.9805
11	1.46	10.7	0.9811
12	1.35	11.6	0.9741
13	1.25	12.6	0.9741
14	1.16	13.6	0.9725
15	1.09	14.4	0.9663
16	1.02	15.5	0.9699
17	0.96	16.4	0.9671
18	0.92	17.0	0.9479
19	0.88	17.9	0.9458
20	0.82	19.1	0.9595
21	0.78	20.1	0.9573
22	0.75	21.0	0.9550
23	0.72	21.8	0.9510
24	0.70	22.6	0.9420
25	0.67	23.5	0.9411
26	0.64	24.4	0.9407
27	0.62	25.1	0.9329
28	0.60	26.0	0.9294
29	0.58	26.9	0.9284
30	0.56	27.8	0.9277

5. Removal of Duplicates

When new tuples are created by projecting nonkey attributes there is always the potential for duplication of tuples. The option of whether to allow duplication of tuples or not will be up the user. If the user allows duplicates then there is nothing extra to do. If duplicates are not desired then a method to remove them must be implemented. There are several different methods available for removing duplicates. Some of these included traversing the file r times and excluding any duplicates, first sorting the file and then removing any duplicates, and using a chain-linked hash table to enter tuples and then discard any duplicates that happen to follow. The hashing method was decided upon because its execution time would be linear with respect to file size. There are three main parts in the

removal of duplicates by hashing algorithm: hashing the tuples into the chain-linked hash table, determining the size of and allocating the output file without duplicates and lastly, unloading the hash table into the output file in parallel.

5.1. Hashing the Tuples

Duplicates will be removed by placing the first occurrence of every tuple into a chained bucket hash table and discarding all subsequent duplicates. A hash table must be allocated and its size should be a function of the number of tuples in the initial file.

If the hash table and all of its buckets had to be allocated on one processor's memory then several problems would develop. As the file size increases, the amount of memory on one processor may not be able to hold all of the tuples, severely restricting the maximum size of a file. Secondly by storing all of the tuples in one processor's memory will cause contention because only one processor can access another processor's memory at a time and seriously degrade performance. It would be nice if we could distribute the rows of the hash table across all of the memories and allow chains of buckets to run from processor to processor. The Butterfly has a mechanism for distributing rows of an array across all the memories of the machine. This mechanism is called a scatter matrix and it will distribute an $M \times N$ array by allocating the M rows to different processor's memory. Each row will be a vector of size N and can be accessed using standard C syntax (A[i][j]). This will reduce contention for each row by a factor of ρ, the number of processors. Also this will increase the maximum size of the file to be operated on by allowing the file to occupy all of the processors' memory. This scatter matrix is maintained by a vector of M pointers each of which points to the next row in the array. This vector is stored on one processor's memory thus causing a secondary place of contention, however, this contention can be removed by making a local copy of this vector at each processor that will be accessing the scatter matrix.

The three fields that make up each slot of the hash table are a pointer to the first bucket (initialized to NULL), a counter field to keep track of the number of tuples in this chain and a lock field (both initialized to 0). The lock field will allow only one processor to manipulate a particular row at a time. Since each row of the hash table has its own lock, only those processors with tuples that have

hashed to that row will have to wait. By choosing a large enough hash table, the number of requests for the same row will be kept to a minimum. Only the code involved with chains will be locked. This is kept to a minimum to prevent processor wait.

Since each new tuple must be compared against those already in the hash table, we would like to keep the length of the chains to a minimum. A simple starting size for the hash table that would keep the chain length very short would be τ, the number of tuples in the file. Not knowing the percentage of duplicates that would be deleted will give only an upper limit on the number of tuples that will be inserted into the hash table. If there are a lot of duplicates that will be removed then most of the rows will be empty. This can be very wasteful if the file size is very large ($>$ 10,000 tuples). Assuming we do not know if any set of the attributes will form a key then we can put an upper limit on the *average* number of tuples in each row. If we would like an average of at most σ tuples to be inserted into each row then the number of rows M should be $M = \tau/\sigma$. Some rows may exceed σ tuples but the average will always be less than or equal to σ. The tuples within a chain will be sorted so that on the average only one-half of the chain will have to be searched to see if it has already been inserted.

The hash function used here is to add up the value of every group of 16 bits in a tuple and return the remainder of this value divided by the table size. This function may be altered if necessary.

5.2. Allocating a New File

Here a new file that is large enough to accommodate the remaining distinct tuples must be created. Each row of the hash table has a count field with the number of tuples in that row. By adding up this count fields will give us the number of tuples in the new file. Now a new file can be statically predeclared with the correct size. Depending on the size of the hash table, the adding up of the count fields can progress in parallel. This step was done serially in this implementation since for small hash tables the overhead needed to start several processors to traverse the table will not save very much time if any at all.

The RAMFile parameters (segment size and the number of segments) will be chosen based upon our experience with the projection operation: the segment size should be no more than $\theta\delta/2$

5.3. Unloading the Hash Table

The task of unloading the hash table into the newly created file will be done by partitioning the hash table into ρ sections. Each of the ρ sections will consist of at most $\lceil M/\rho \rceil$ rows of the hash table. Every task will add up the total amount of tuples in its block of rows, then it will atomically retrieve and increment a EOF pointer by the number of tuples in its block. This atomic operation will return the current EOF position in the new file and increment it by the number of tuples in its range thus reserving enough space to accommodate all of its tuples. Finally the task will loop through its set of rows and write each tuple into the new file releasing each bucket in turn.

5.4. Removal of Duplicates Results

The removal of duplicates algorithm was implemented and timed using 1 through 30 equivalent processors on the Butterfly and the relations with 10,000, 1,000 and 100 tuples. For each distinct relation size, we selected 3 unique relations containing 100%, 50% and 10% distinct tuples, respectively. Thus for each distinct relation there will be a 0%, 50% and 90% reduction in the size of the output relation. Input and output relations will have RAMFile parameters as developed in the projection section. The results of performing the removal of duplicates algorithm can be seen in Tables 23 - 25.

Removing duplicates from the larger relations was the most efficient, with efficiency decreasing as the number of distinct tuples decreases. There are several important reasons for this observation. The main reason for the drop in efficiency as the size of the input relation decreases is because there is less work for each processor to do and the overhead necessary to initialize and start several smaller tasks becomes a larger percentage of the actual execution time. Another problem occurs when working on smaller relations. Suppose we divide 100 tuples into blocks of 5 tuples. This will create 20 tasks that can be operated on by 20 different processors. Using 21, 22, 23 or 24 processors will not allow any more parallelization since all 20 tasks will have been allocated. Not until we use 25 processors will the task size will be reduced to 4 tuples. From here (25 tasks) no additional tasks are created

Table 23 10,000 Tuple Removal of Duplicates						
	10,000 Distinct		5,000 Distinct		1,000 Distinct	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	87.28	1.0	60.63	1.0	27.78	1.0
2	49.32	1.7	33.55	1.8	17.76	1.5
3	37.93	2.3	23.33	2.5	13.17	2.1
4	31.09	2.8	17.38	3.4	10.50	2.6
5	26.52	3.2	13.24	4.5	8.75	3.1
6	21.37	4.0	11.03	5.4	7.67	3.6
7	17.82	4.8	9.49	6.3	6.87	4.0
8	14.08	6.1	8.28	7.3	6.42	4.3
9	12.56	6.9	7.44	8.1	5.73	4.8
10	10.96	7.9	6.99	8.6	5.36	5.1
11	9.47	9.2	6.19	9.7	5.09	5.4
12	8.56	10.1	5.82	10.4	4.80	5.7
13	7.99	10.9	5.35	11.3	4.54	6.1
14	7.23	12.0	5.01	12.1	4.34	6.3
15	6.64	13.1	4.76	12.7	4.23	6.5
16	6.31	13.8	4.68	12.9	4.01	6.9
17	5.91	14.7	4.48	13.5	4.01	6.9
18	5.61	15.5	4.31	14.0	3.80	7.3
19	5.29	16.4	4.09	14.7	3.78	7.3
20	5.11	17.0	4.10	14.7	3.66	7.5
21	4.97	17.5	3.95	15.3	3.58	7.7
22	4.69	18.5	3.84	15.7	3.49	7.9
23	4.53	19.2	3.81	15.8	3.47	8.0
24	4.38	19.9	3.76	16.1	3.41	8.1
25	4.32	20.1	3.66	16.5	3.36	8.2
26	4.13	21.1	3.61	16.7	3.32	8.3
27	4.05	21.5	3.52	17.1	3.27	8.4
28	3.98	21.9	3.50	17.3	3.26	8.5
29	3.94	22.1	3.48	17.3	3.21	8.6
30	3.85	22.6	3.40	17.7	3.18	8.7

until we use 34 processors, giving us a block size of 3 tuples. This particular problem is only relevant when the number of tuples approaches the number of processors.

The second major problem concerns the RAMFile allocation and creation stage. As pointed out previously, RAMFile creation is a serial operation that must know ahead of time how much memory to reserve for the new file. Since the file must be allocated in several segments, the time necessary to allocate each segment becomes a major contributor to the execution time as the number of processors increases. We have placed an lower limit of 2 segments per processor to allow a buffer of one segment between two consecutive tasks. (A more detailed formula which allows the computation of the correct

Table 24 1,000 Tuple Removal of Duplicates						
	1,000 Distinct		500 Distinct		100 Distinct	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	4.66	1.0	4.14	1.0	3.76	1.0
2	3.21	1.4	2.82	1.4	2.56	1.4
3	2.44	1.9	2.18	1.8	2.03	1.8
4	2.05	2.2	1.87	2.2	1.77	2.1
5	1.82	2.5	1.71	2.4	1.56	2.4
6	1.67	2.7	1.58	2.6	1.48	2.5
7	1.59	2.9	1.54	2.6	1.45	2.5
8	1.51	3.0	1.47	2.8	1.36	2.7
9	1.49	3.1	1.43	2.8	1.33	2.8
10	1.43	3.2	1.40	2.9	1.32	2.8
11	1.40	3.3	1.39	2.9	1.29	2.9
12	1.37	3.3	1.40	2.9	1.29	2.9
13	1.38	3.3	1.37	3.0	1.27	2.9
14	1.41	3.3	1.38	2.9	1.26	2.9
15	1.39	3.3	1.38	2.9	1.26	2.9
16	1.38	3.3	1.37	3.0	1.23	3.0
17	1.37	3.3	1.37	3.0	1.23	3.0
18	1.37	3.3	1.35	3.0	1.22	3.0
19	1.37	3.3	1.37	3.0	1.20	3.1
20	1.34	3.4	1.36	3.0	1.24	3.0
21	1.34	3.4	1.34	3.0	1.24	3.0
22	1.33	3.4	1.35	3.0	1.23	3.0
23	1.35	3.4	1.36	3.0	1.22	3.0
24	1.32	3.5	1.37	3.0	1.22	3.0
25	1.31	3.5	1.38	3.0	1.24	3.0
26	1.36	3.4	1.34	3.0	1.26	2.9
27	1.38	3.3	1.38	2.9	1.26	2.9
28	1.40	3.3	1.37	3.0	1.29	2.9
29	1.39	3.3	1.41	2.9	1.28	2.9
30	1.41	3.3	1.41	2.9	1.26	2.9

segment size will be developed below.)

Although the time necessary to allocate a RAMFile may only be a small percentage of the execution time when only one processor is being utilized, this is not so when several processors are involved. For each timing of the algorithm using varying relation sizes and number of processors, intermediate times were measured to observe how long it was taking to allocate the new RAMFiles serially. From these intermediate times it was verified that the time to allocate a RAMFile was a constant that did not vary with the number of processors. To demonstrate the effect of not having to predeclare the size of the new RAMFile we deducted in Table 26 the time necessary to allocate the file

Table 25 100 Tuple Removal of Duplicates						
	100 Distinct		50 Distinct		10 Distinct	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	1.22	1.0	0.86	1.0	0.59	1.0
2	0.98	1.2	0.71	1.2	0.45	1.2
3	0.90	1.3	0.60	1.4	0.37	1.5
4	0.86	1.4	0.58	1.4	0.35	1.6
5	0.83	1.4	0.56	1.5	0.33	1.7
6	0.84	1.4	0.55	1.5	0.32	1.8
7	0.88	1.3	0.58	1.4	0.34	1.7
8	0.88	1.3	0.60	1.4	0.36	1.6
9	0.89	1.3	0.62	1.3	0.37	1.5
10	0.91	1.3	0.61	1.4	0.36	1.5
11	0.96	1.2	0.65	1.3	0.38	1.5
12	0.94	1.3	0.64	1.3	0.38	1.5
13	0.94	1.2	0.65	1.3	0.38	1.5
14	0.98	1.2	0.67	1.2	0.41	1.4
15	0.97	1.2	0.66	1.2	0.40	1.4
16	1.00	1.2	0.68	1.2	0.41	1.4
17	0.98	1.2	0.70	1.2	0.42	1.4
18	1.02	1.2	0.71	1.2	0.42	1.3
19	1.04	1.1	0.67	1.2	0.45	1.2
20	1.00	1.2	0.71	1.2	0.44	1.3
21	1.02	1.1	0.72	1.1	0.46	1.2
22	1.03	1.1	0.73	1.1	0.49	1.1
23	1.08	1.1	0.77	1.1	0.50	1.1
24	1.06	1.1	0.77	1.1	0.47	1.2
25	1.05	1.1	0.75	1.1	0.57	1.0
26	1.08	1.1	0.78	1.1	0.53	1.1
27	1.06	1.1	0.79	1.0	0.56	1.0
28	1.10	1.1	0.81	1.0	0.57	1.0
29	1.08	1.1	0.82	1.0	0.56	1.0
30	1.09	1.1	0.79	1.0	0.58	1.0

from the execution times for a relation that contains 10,000 distinct tuples and recalculated the effective speedup, placing the additional results in the last two columns. As can be seen the difference between effective processors increases as the number of processors increases because the file creation time becomes a larger percentage of the total execution time. If there were some way to reduce the time it takes to allocate the new RAMFile it would greatly improve the execution times for all sets of execution parameters especially those with the smaller relation sizes.

Table 26 10,000 Tuple Removal of Duplicates 10,000 Distinct Tuples				
	With RAMFile Creation Time		Without RAMFile Creation Time: 0.91 sec	
No. of Processors	Time	Effective Processors	Time	Effective Processors
1	87.28	1.0	86.37	1.0
2	49.32	1.7	48.41	1.8
3	37.93	2.3	37.02	2.3
4	31.09	2.8	30.18	2.9
5	26.52	3.2	25.61	3.4
6	21.37	4.0	20.46	4.2
7	17.82	4.8	16.91	5.1
8	14.08	6.1	13.17	6.6
9	12.56	6.9	11.65	7.4
10	10.96	7.9	10.05	8.6
11	9.47	9.2	8.56	10.1
12	8.56	10.1	7.65	11.3
13	7.99	10.9	7.08	12.2
14	7.23	12.0	6.32	13.7
15	6.64	13.1	5.73	15.1
16	6.31	13.8	5.40	16.0
17	5.91	14.7	5.00	17.3
18	5.61	15.5	4.70	18.4
19	5.29	16.4	4.38	19.7
20	5.11	17.0	4.20	20.6
21	4.97	17.5	4.06	21.3
22	4.69	18.5	3.78	22.8
23	4.53	19.2	3.62	23.9
24	4.38	19.9	3.47	24.9
25	4.32	20.1	3.41	25.3
26	4.13	21.1	3.22	26.8
27	4.05	21.5	3.14	27.5
28	3.98	21.9	3.07	28.1
29	3.94	22.1	3.03	28.5
30	3.85	22.6	2.94	29.4

6. The Selection Operation

Two methods for performing the selection operation will be demonstrated below and analyzed. Similar to the projection operation, the selection algorithm will entail file division, selection of tuples and reunion of output. Since the output file must be statically predeclared to the proper size, we will require an extra step of computing the number of tuples being selected and allocating a large enough output file so that they can be inserted into it. We will look at two ways to accomplish this, either by saving all of the selected tuples in a table (memory permitting) or just keeping a list of the positions of the selected tuples.

In performing the selection operation, there are several things that can be incorporated from the project operation. These will include the methods of file division and RAMFile distribution parameters. From the removal of duplicates algorithm will be the use of a hash table scattered across all of the processors' memories. Also borrowed from the removal of duplicates algorithm will be the method of computing the output file size and output file creation.

6.1. File Division

The relation will be divided up among all of the available processors in equal sized blocks as with the projection operation. Initially we will assume that all processors are executing at approximately the same speed, so for the initial block size δ we will start with $\delta = \lceil \tau/\rho \rceil$

6.2. Selection of Tuples

6.2.1. Selected Address Table Method

Depending on the amount of main memory or the number of processors available will determine which of the following two algorithms will be used. The first algorithm will *not* store the actual tuples after selecting them, rather it will just remember the position of the tuple within the input file and increment a tuple counter. Each task will scan its block of tuples and remember the positions of the tuples that are selected in a global array of integers. Each successive newly selected tuple will be placed in the next adjacent slot in the table. The end of the table will be remembered by a global EOT (end of table) pointer. The incrementing of this EOT pointer will have to be an atomic operation.

With the projection operation the EOF pointer was unsatisfactory since it serialized the actual write operations to the output file and because several tuples resided in one segment this caused a backlog of writes, severely degrading performance. With the EOT pointer, however, the only serial code is the actual atomic incrementing of the pointer. This operation takes approximately 50 microseconds and the only transfer of data will be just an integer transfer which is only four bytes in length.

The major drawback with using a selected address table is that all tuples that were selected will

have to be reread back into memory again so that they can be written back out to the output file. It takes approximately 7.9 seconds to read in 10,000 tuples on one processor, so if the number of selected tuples approaches the number of tuples in the relation you could be adding approximately 33% to the I/O time of the selection.

6.2.2. Hash Table Method

In order to avoid reading selected tuples into memory twice, we would like to be able to save all of the selected tuples in main memory while the new output file was being sized and created. Also we would like to keep this very dynamic so that only enough main memory to hold the selected tuples will be allocated. Using dynamic memory allocation in conjunction with several linked lists will allow us to only allocated only the memory that we need and an easy way to retrieve it. Each task will have its own linked list to prevent contention for the root pointer. All of the linked lists will be stored together in a global scatter matrix to prevent contention also. Associated with each linked list will be a counter variable to account for the number of tuples in each linked list.

Each task will read in its tuples and if it fulfills the selection requirements will be inserted into the front of its linked list. Next the counter associated with this list will be atomically incremented to account for the new tuple. A possible problem here is the fact that each linked list will not be the same size as any of the others. This will be discussed later.

6.3. Reunion of Output

6.3.1. Selected Address Table Method

Before any tuples can be placed into the output file, the file's size must first be determined and then allocated. This step is a problem because file creation is a serial operation and its execution time is relative to the size of the new file. Since the global EOT pointer contains the index into the table of the last tuple to be selected we know how many tuples have been selected. Some file creation times are shown in Table 27 with their sizes and RAMFile parameters.

As is evident from Table 27 the file creation time is mostly dependent on the number of segments in the file as compared to total file size. Taking this into consideration we would like to minimize the

Table 27 Time to Create RAMFiles			
No. of Segments	Segment Size	File Size (Bytes)	Time (Secs)
50	256	12800	0.388
50	512	25600	0.387
50	1024	51200	0.379
50	2048	102400	0.396
50	4096	204800	0.383
100	256	25600	0.737
100	512	51200	0.731
100	1024	102400	0.740
100	2048	204800	0.737
100	4096	409600	0.758
150	256	38400	1.119
150	512	76800	1.082
150	1024	153600	1.098
150	2048	307200	1.106
150	4096	614400	1.100
200	256	51200	1.444
200	512	102400	1.464
200	1024	204800	1.459
200	2048	409600	1.470
200	4096	819200	1.459
250	256	64000	1.810
250	512	128000	1.807
250	1024	256000	1.798
250	2048	512000	1.816
250	4096	1024000	1.822
300	256	76800	2.161
300	512	153600	2.145
300	1024	307200	2.150
300	2048	614400	2.151
300	4096	1228800	2.169
350	256	89600	2.523
350	512	179200	2.511
350	1024	358400	2.519
350	2048	716800	2.512
350	4096	1433600	2.540
400	256	102400	2.887
400	512	204800	2.893
400	1024	409600	2.885
400	2048	819200	2.859
400	4096	1638400	2.916
450	256	115200	3.256
450	512	230400	3.243
450	1024	460800	3.237
450	2048	921600	3.208
450	4096	1843200	3.252

number of segments in the newly created file. The problem with this is if there are too few segments to distribute the file between contention will occur. One idea would be to try to keep output from each task localized so that it would be scattered over as few segments as possible. Optimally then we would like to allocate as many segments as there are processors. Usually this can be attained although the RAMFile has certain limitations such as the maximum segment size is 64 Kbytes and that segments can only be a power of two in length. So if the file size is greater than $\rho \times 64$ Kbytes long, then you will have to allocate more segments. Another problem is that if $\tau\theta/\rho$ is not a power of two then each task's block of tuples will overlap onto another processor's segment possible causing contention. To prevent block overlap we would like each task's block to be distributed across at least 2 or 3 segments. This is so that for any two contiguous blocks there should always be a buffer of at least 1 or 2 segments between them. Let α represent the number of segments in a RAMFile and β the segment size (remember beta is equal to 2^{x} $(8 \leq x \leq 16)$ then a formula for the segment size would now be:

$$\beta = 2^{\left\lfloor \log_2\left(\frac{\delta}{2}\right)\right\rfloor}$$

with the number of segments being:

$$\alpha = \left\lceil \frac{\tau\theta}{\beta} \right\rceil$$

Once the file has been created the tuples that were remembered in the selection address table must be read back into memory and then placed into the newly created output file. To achieve maximum parallelization we will distribute the selected tuple addresses evenly between all processors. We will assume all processors are executing at similar speeds, thus each task will be given $\left\lceil \frac{\textit{no. selected tuples}}{\rho} \right\rceil$ tuples to get and write. If any speed differences are noticed the equations used to compute new block sizes from the projection routine will be used. Since each task will know how big each block size is and which block it is it will be possible for each task to place all of its selected tuples in one continuous block in the output file.

6.3.2. Hash Table Method

The only difference for the file creation stage between the two algorithms is that for the hash table method the total tuple count must be performed. This is a relatively simple task of adding up all of the linked list count variables. Depending on how many variables and linked list there are you may want to tally these in parallel. For our purposes since there are only 30 processors it will be done serially.

As was noted above each linked list may have an arbitrary number of tuples stored in it. Hence it will not be as easy to compute where each task's block of tuples will be placed in the output file. What we will do is initialize a EOF pointer to the beginning of the file and have each task read this value and then increment it by the number of tuples in its linked list. The read and increment must be performed atomically so a lock must be placed around this section of the code. This will retrieve the start of the task's output block and reserve it enough space to hold all of its selected tuples. The only thing remaining to be performed is to unload the linked list into its own block of the output file, freeing the tuples dynamic memory as they are written out.

6.4. Selection Results

Since most of the development of the RAMFile parameters was done for the projection operation we will start off where the project operation left off. This means that the input relations will have segment sizes 16 kbytes, 2 kbytes and 256 bytes for the 10,000, 1,000 and the 100 tuple relations, respectively. For each relation the number of tuples being selected will be: 100%, 50%, 10% and 1. First the tables showing the results of the selected address table method (Tables 28 - 30) will be displayed followed by the hash table method (Tables 31 - 33).

The results of both algorithms were very comparable in execution time for most of the test cases, the only variations being that the hash table method was slightly faster than the selected address method when run on only a few (1 - 5) processors and when the size of the input and output relations was very large (10,000 tuples in and 10,000 and 5,000 tuples out). Only for the larger relations was the difference significant. Since most of the times were very close between matching param-

Table 28 10,000 Tuple Selection Selected Address Table Method								
Selected Tuples	10,000 Tuples		5,000 Tuples		1,000 Tuples		1 Tuple	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	27.38	1.0	18.13	1.0	10.55	1.0	7.70	1.0
2	14.42	1.8	9.66	1.8	5.70	1.8	3.93	1.9
3	10.15	2.6	6.92	2.6	4.15	2.5	2.68	2.8
4	10.36	2.6	6.09	2.9	3.39	3.1	2.06	3.7
5	9.61	2.8	5.45	3.3	2.94	3.5	1.70	4.5
6	8.35	3.2	4.98	3.6	2.62	4.0	1.44	5.3
7	7.48	3.6	4.56	3.9	2.41	4.3	1.28	6.0
8	6.76	4.0	4.18	4.3	2.23	4.7	1.15	6.6
9	6.25	4.3	3.94	4.5	2.10	5.0	1.08	7.1
10	5.87	4.6	3.73	4.8	1.98	5.3	0.99	7.7
11	5.53	4.9	3.51	5.1	1.90	5.5	0.93	8.2
12	5.22	5.2	3.35	5.4	1.85	5.6	0.89	8.5
13	5.00	5.4	3.21	5.6	1.78	5.9	0.83	9.1
14	4.77	5.7	3.10	5.8	1.75	6.0	0.81	9.4
15	4.66	5.8	3.02	6.0	1.69	6.2	0.78	9.7
16	4.49	6.0	2.90	6.2	1.68	6.2	0.77	9.9
17	4.41	6.1	2.86	6.3	1.63	6.4	0.74	10.3
18	4.33	6.3	2.80	6.4	1.61	6.5	0.74	10.3
19	4.19	6.5	2.77	6.5	1.59	6.6	0.71	10.8
20	4.21	6.4	2.71	6.6	1.56	6.7	0.70	10.8
21	4.20	6.5	2.66	6.8	1.58	6.6	0.68	11.1
22	4.14	6.6	2.67	6.7	1.59	6.6	0.71	10.7
23	4.21	6.4	2.66	6.8	1.58	6.6	0.74	10.4
24	4.29	6.3	2.63	6.8	1.53	6.8	0.70	10.9
25	4.38	6.2	2.66	6.8	1.55	6.7	0.71	10.7
26	4.48	6.1	2.70	6.7	1.52	6.9	0.72	10.5
27	4.50	6.0	2.71	6.6	1.52	6.9	0.71	10.7
28	4.68	5.8	2.71	6.6	1.55	6.7	0.73	10.5
29	4.85	5.6	2.84	6.3	1.55	6.7	0.72	10.5
30	4.99	5.4	2.85	6.3	1.53	6.8	0.71	10.7

eters (except for the cases noted above) it may allow us to make some rough assumptions about the time it takes to allocate and free dynamic memory and RAMFile I/O time.

Both algorithms were very similar except that the selected address method must read in all tuples once and all selected tuples twice, compared to only once for all tuples in the hash table method. What makes the hash table method different is that for every selected tuple memory large enough to hold each selected tuple must be dynamically allocated when it is read in and freed up when it is written out to the output relation. This means that every selected tuple must be read in an extra time in the selected address method compared to having to allocate and free up dynamic

Table 29 1,000 Tuple Selection Selected Address Table Method								
Selected Tuples	1,000 Tuples		500 Tuples		100 Tuples		1 Tuple	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	3.86	1.0	2.92	1.0	1.97	1.0	0.99	1.0
2	2.35	1.6	1.87	1.5	1.30	1.5	0.55	1.7
3	1.91	2.0	1.53	1.8	1.09	1.7	0.41	2.3
4	1.81	2.1	1.40	2.0	1.01	1.9	0.32	3.0
5	1.69	2.2	1.34	2.1	0.92	2.1	0.29	3.3
6	1.59	2.4	1.26	2.3	0.91	2.1	0.28	3.5
7	1.49	2.5	1.23	2.3	0.90	2.1	0.24	4.0
8	1.44	2.6	1.19	2.4	0.90	2.1	0.24	4.0
9	1.40	2.7	1.16	2.5	0.88	2.2	0.24	4.0
10	1.34	2.8	1.15	2.5	0.87	2.2	0.24	4.0
11	1.33	2.9	1.13	2.5	0.90	2.1	0.23	4.1
12	1.31	2.9	1.13	2.5	0.88	2.2	0.24	4.0
13	1.28	3.0	1.16	2.5	0.89	2.2	0.24	3.9
14	1.27	3.0	1.12	2.5	0.90	2.1	0.26	3.7
15	1.26	3.0	1.10	2.6	0.90	2.1	0.28	3.4
16	1.27	3.0	1.10	2.6	0.89	2.1	0.28	3.4
17	1.28	3.0	1.11	2.6	0.90	2.1	0.28	3.5
18	1.28	3.0	1.10	2.6	0.91	2.1	0.30	3.2
19	1.26	3.0	1.11	2.6	0.93	2.1	0.32	3.0
20	1.26	3.0	1.13	2.5	0.90	2.1	0.32	3.0
21	1.27	3.0	1.15	2.5	0.94	2.0	0.32	3.0
22	1.29	2.9	1.14	2.5	0.98	1.9	0.31	3.1
23	1.30	2.9	1.18	2.4	0.96	2.0	0.35	2.7
24	1.34	2.8	1.16	2.5	0.97	2.0	0.34	2.9
25	1.30	2.9	1.18	2.4	0.99	1.9	0.39	2.5
26	1.37	2.8	1.18	2.4	1.01	1.9	0.38	2.5
27	1.33	2.9	1.17	2.4	1.00	1.9	0.40	2.4
28	1.37	2.8	1.20	2.4	1.04	1.8	0.43	2.2
29	1.40	2.7	1.24	2.3	1.03	1.8	0.43	2.2
30	1.41	2.7	1.23	2.3	1.05	1.8	0.42	2.3

memory to store it in the hash table method. Since most execution times were very close leads us to assume that RAMFile retrieval times may be very close if not slightly longer than the time necessary for dynamic memory allocation and releasing.

Another reoccurring problem is the serial RAMFile creation time. This was first noticed in the removal of duplicates section and seems to be a necessary delay until the RAMFile system software can be updated and improved. One such improvement could include a more dynamic memory allocation routine during file creation. This would allow memory to be allocated for a segment in a RAMFile when and only if the segment was about to be written to, otherwise that segment would not exist

Table 30 100 Tuple Selection Selected Address Table Method								
Selected Tuples	100 Tuples		50 Tuples		10 Tuples		1 Tuple	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	1.30	1.0	0.81	1.0	0.40	1.0	0.28	1.0
2	0.99	1.3	0.61	1.3	0.27	1.4	0.19	1.4
3	0.90	1.4	0.52	1.5	0.22	1.7	0.17	1.6
4	0.85	1.5	0.50	1.6	0.22	1.8	0.14	1.9
5	0.83	1.5	0.49	1.6	0.21	1.8	0.16	1.7
6	0.82	1.5	0.51	1.6	0.22	1.7	0.16	1.6
7	0.82	1.5	0.50	1.6	0.25	1.5	0.20	1.4
8	0.81	1.5	0.53	1.5	0.25	1.5	0.18	1.5
9	0.83	1.5	0.51	1.5	0.27	1.4	0.17	1.5
10	0.82	1.5	0.52	1.5	0.28	1.4	0.21	1.3
11	0.87	1.4	0.53	1.5	0.26	1.5	0.20	1.4
12	0.84	1.5	0.54	1.5	0.30	1.3	0.21	1.2
13	0.84	1.5	0.54	1.5	0.32	1.2	0.22	1.2
14	0.87	1.4	0.56	1.4	0.29	1.3	0.21	1.2
15	0.89	1.4	0.57	1.4	0.31	1.3	0.24	1.1
16	0.90	1.4	0.59	1.3	0.31	1.2	0.23	1.2
17	0.89	1.4	0.58	1.4	0.30	1.3	0.26	1.0
18	0.88	1.4	0.57	1.4	0.34	1.1	0.26	1.0
19	0.92	1.4	0.59	1.3	0.33	1.2	0.28	0.9
20	0.91	1.4	0.61	1.3	0.35	1.1	0.28	0.9
21	0.94	1.3	0.63	1.2	0.34	1.1	0.27	1.0
22	0.94	1.3	0.61	1.3	0.39	1.0	0.32	0.8
23	0.97	1.3	0.66	1.2	0.39	1.0	0.30	0.9
24	0.96	1.3	0.67	1.2	0.40	0.9	0.34	0.8
25	1.00	1.2	0.67	1.2	0.40	0.9	0.32	0.8
26	0.98	1.3	0.71	1.1	0.45	0.8	0.35	0.7
27	0.98	1.3	0.71	1.1	0.43	0.9	0.32	0.8
28	1.00	1.2	0.75	1.0	0.44	0.9	0.39	0.7
29	1.04	1.2	0.73	1.1	0.47	0.8	0.42	0.6
30	1.02	1.2	0.74	1.1	0.44	0.9	0.41	0.6

in main memory and would return zero values for any read operations that may occur before any write took place. By postponing the memory allocation until it was needed would reduce the initial RAMFile creation time. For each run of the algorithm intermediate times were taken to time crucial portions of the code such as when the new RAMFile was created. The intermediate times to create the RAMFiles were very similar to those in the removal of duplicates for corresponding file sizes. These times are directly proportional to the number of segments in the new file and not related to the size of the segment at all. Thus times may be reduced by decreasing the number of segments by increasing the size of each segment. This seems like a good idea until we realize that we may be

Table 31 10,000 Tuple Selection Hash Table Method								
Selected Tuples	10,000 Tuples		5,000 Tuples		1,000 Tuples		1 Tuple	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	26.51	1.0	17.37	1.0	10.09	1.0	7.65	1.0
2	14.04	1.8	9.54	1.8	5.60	1.8	3.90	1.9
3	10.06	2.6	6.71	2.5	4.03	2.5	2.67	2.8
4	7.84	3.3	5.30	3.2	3.21	3.1	2.05	3.7
5	6.37	4.1	4.48	3.8	2.78	3.6	1.67	4.5
6	5.53	4.7	3.98	4.3	2.47	4.0	1.43	5.3
7	4.92	5.3	3.59	4.8	2.27	4.4	1.28	5.9
8	4.45	5.9	3.29	5.2	2.12	4.7	1.14	6.6
9	4.10	6.4	3.07	5.6	2.01	5.0	1.06	7.1
10	3.85	6.8	2.86	6.0	1.88	5.3	0.98	7.7
11	3.59	7.3	2.73	6.3	1.82	5.5	0.91	8.3
12	3.42	7.7	2.60	6.6	1.75	5.7	0.88	8.6
13	3.27	8.0	2.49	6.9	1.72	5.8	0.83	9.1
14	3.12	8.4	2.42	7.1	1.66	6.0	0.80	9.5
15	3.03	8.7	2.37	7.3	1.65	6.1	0.78	9.7
16	2.95	8.9	2.30	7.5	1.61	6.2	0.77	9.8
17	2.87	9.2	2.23	7.7	1.60	6.2	0.74	10.2
18	2.76	9.5	2.23	7.7	1.55	6.4	0.71	10.6
19	2.73	9.6	2.17	7.9	1.56	6.4	0.72	10.6
20	2.67	9.8	2.12	8.1	1.52	6.6	0.72	10.5
21	2.63	10.0	2.10	8.2	1.51	6.6	0.71	10.6
22	2.62	10.1	2.08	8.3	1.54	6.5	0.69	10.9
23	2.57	10.2	2.06	8.4	1.51	6.6	0.71	10.6
24	2.53	10.4	2.05	8.4	1.50	6.7	0.71	10.6
25	2.50	10.5	2.05	8.4	1.47	6.8	0.71	10.7
26	2.49	10.6	2.01	8.6	1.48	6.8	0.68	11.1
27	2.42	10.9	2.02	8.5	1.49	6.7	0.71	10.6
28	2.46	10.7	2.00	8.6	1.50	6.7	0.69	11.0
29	2.46	10.7	2.01	8.6	1.49	6.7	0.73	10.4
30	2.43	10.8	1.98	8.7	1.49	6.7	0.72	10.5

creating contention problems by squeezing too may tasks' tuples into one segment.

7. The Scalar Aggregate Operations

The calculation of the scalar aggregate operation will be very similar to the first part of the projection operation. Here again the file will be divided into smaller blocks, then each task will pass through its block performing the designated operation (maximum, minimum, average or count) on its block of data only. At the end, each task will assimilate its local answer into a global response.

Table 32 1,000 Tuple Selection Hash Table Method								
Selected Tuples	1,000 Tuples		500 Tuples		100 Tuples		1 Tuple	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	3.46	1.0	2.63	1.0	1.85	1.0	1.00	1.0
2	2.20	1.5	1.75	1.5	1.25	1.4	0.53	1.8
3	1.73	1.9	1.45	1.8	1.08	1.7	0.39	2.5
4	1.51	2.2	1.28	2.0	0.98	1.8	0.32	3.0
5	1.37	2.5	1.18	2.2	0.95	1.9	0.28	3.5
6	1.32	2.6	1.15	2.2	0.91	2.0	0.25	3.8
7	1.27	2.7	1.12	2.3	0.90	2.0	0.25	3.9
8	1.23	2.8	1.11	2.3	0.91	2.0	0.24	4.0
9	1.20	2.8	1.11	2.3	0.93	1.9	0.24	4.1
10	1.19	2.8	1.11	2.3	0.93	1.9	0.24	4.1
11	1.18	2.9	1.12	2.3	0.93	1.9	0.25	3.9
12	1.18	2.9	1.12	2.3	0.94	1.9	0.25	3.9
13	1.17	2.9	1.13	2.3	0.95	1.9	0.24	4.0
14	1.17	2.9	1.14	2.3	0.94	1.9	0.25	3.9
15	1.17	2.9	1.11	2.3	0.97	1.8	0.24	4.1
16	1.19	2.8	1.14	2.2	0.97	1.8	0.27	3.6
17	1.18	2.9	1.14	2.3	0.95	1.9	0.28	3.5
18	1.19	2.8	1.13	2.3	0.96	1.9	0.29	3.3
19	1.18	2.9	1.13	2.3	0.97	1.9	0.30	3.2
20	1.17	2.9	1.15	2.2	0.96	1.9	0.29	3.4
21	1.16	2.9	1.15	2.2	0.97	1.9	0.33	2.9
22	1.18	2.9	1.13	2.3	0.96	1.9	0.33	2.9
23	1.20	2.8	1.17	2.2	1.02	1.8	0.34	2.9
24	1.21	2.8	1.16	2.2	0.98	1.8	0.32	3.1
25	1.21	2.8	1.14	2.2	0.99	1.8	0.34	2.8
26	1.23	2.8	1.18	2.2	1.00	1.8	0.33	2.9
27	1.22	2.8	1.20	2.1	1.01	1.8	0.39	2.5
28	1.22	2.8	1.19	2.2	1.02	1.8	0.36	2.7
29	1.22	2.8	1.21	2.1	1.06	1.7	0.42	2.3
30	1.28	2.7	1.21	2.1	1.04	1.7	0.42	2.3

7.1. File Division

As improvements have developed they have been incorporated into successive operations. Thus the same algorithm used to compute the optimal block size developed in the projection routine and used in the selection routine will be incorporated here. Initially we will use a δ of:

$$\delta = \left\lceil \frac{\tau - \left\lfloor \phi \left\lceil \frac{\tau}{\rho} \right\rceil \right\rfloor}{\rho - 1} \right\rceil$$

$$with\ \phi = 1.0$$

Table 33 100 Tuple Selection Hash Table Method								
Selected Tuples	100 Tuples		50 Tuples		10 Tuples		1 Tuple	
No. of Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors	Time	Effective Processors
1	1.15	1.0	0.71	1.0	0.37	1.0	0.29	1.0
2	0.90	1.2	0.54	1.3	0.29	1.2	0.18	1.6
3	0.83	1.3	0.47	1.4	0.23	1.5	0.17	1.7
4	0.81	1.4	0.46	1.5	0.23	1.5	0.15	1.8
5	0.79	1.4	0.46	1.5	0.21	1.7	0.17	1.7
6	0.79	1.4	0.46	1.5	0.21	1.7	0.15	1.9
7	0.81	1.4	0.50	1.4	0.26	1.4	0.19	1.4
8	0.80	1.4	0.52	1.3	0.26	1.4	0.15	1.8
9	0.82	1.4	0.54	1.3	0.26	1.4	0.20	1.4
10	0.82	1.3	0.54	1.3	0.25	1.4	0.17	1.6
11	0.88	1.3	0.57	1.2	0.24	1.5	0.18	1.6
12	0.88	1.3	0.56	1.2	0.27	1.3	0.20	1.4
13	0.90	1.2	0.55	1.2	0.30	1.2	0.20	1.4
14	0.89	1.2	0.59	1.1	0.31	1.1	0.21	1.4
15	0.92	1.2	0.61	1.1	0.29	1.2	0.22	1.3
16	0.93	1.2	0.62	1.1	0.32	1.1	0.22	1.3
17	0.92	1.2	0.58	1.2	0.32	1.1	0.24	1.2
18	0.97	1.1	0.60	1.1	0.31	1.2	0.25	1.1
19	0.97	1.1	0.62	1.1	0.32	1.1	0.24	1.2
20	0.93	1.2	0.65	1.0	0.36	1.0	0.26	1.1
21	0.91	1.2	0.68	1.0	0.34	1.0	0.27	1.0
22	1.00	1.1	0.64	1.1	0.34	1.0	0.30	0.9
23	0.98	1.1	0.70	1.0	0.41	0.9	0.28	1.0
24	0.99	1.1	0.71	1.0	0.38	0.9	0.34	0.8
25	0.96	1.2	0.70	1.0	0.42	0.8	0.30	0.9
26	0.99	1.1	0.70	1.0	0.43	0.8	0.34	0.8
27	1.03	1.1	0.69	1.0	0.37	0.9	0.33	0.8
28	1.02	1.1	0.76	0.9	0.47	0.7	0.36	0.8
29	1.04	1.1	0.72	0.9	0.46	0.8	0.33	0.9
30	1.08	1.0	0.75	0.9	0.45	0.8	0.40	0.7

for our starting block size. As the results progress we will incorporate the same speed ratio modification as used before if necessary.

A way to pass several parameters to the tasks and a method for the tasks to incorporate their results with the global result must be included. This is because all processes will need several parameters to allow them to know the dynamic variables such as input file name, input file size, scalar aggregate operation and a place for the results.

Here we will create a data object that will hold the parameters for the child processes and also space to return their results in. The initialization of this data object will be done serially by the

parent process and then the object identification (OID) number of the data object must be passed to the child processes. Each process that wants to access this data object must first map that object into memory it can access. From this point on, the data object can be accessed and altered by all processes that have mapped in its OID. Since all processes will be accessing the data object, possible inconsistencies may occur.

The only place where there will be opportunity for inconsistency problems is when the children try to incorporate their individual results with the global result. There will not be any other consistency problems spots, because when the parent process is initializing the data object there will be no other processes to contend with and when the child processes are unpacking the parameters, they will only be reading from the data object. Inconsistency will be prevented by placing a lock around the code that will access the global result. Lastly, when a process is done accessing a particular data object, it must unmap the object from its memory because it is consuming a segment attribute register (SAR). SARs are system registers that maintain the location of a data object in memory. In the current system each processor has 512 SARs.

7.2. Aggregation of Tuples

After each child has unpacked its parameters from the data object, they will perform the scalar aggregate function on their own block of tuples. The child's local result will be stored in a private copy of the result. The global result has been initialized to a value that will be replaced immediately with real values (i.e. for a maximum operation an initial value of -9,999,999,999). Thus giving the child task something to compare against initially.

7.3. Reunion of Output

When each child process has completed computing the scalar aggregate function on its own block of tuples, it will have to incorporate its local result into the global result. Each child process will have mapped in the data object that contains the global result so that it will atomically access the global result and either replace it with its result (i.e. if its maximum is greater that the global maximum or add its tuple count and sum for an average to the global sums). A lock around this portion

of the code will prevent data inconsistencies.

7.4. Scalar Aggregate Results

Following the examples of the above operations the execution of the scalar aggregate function will be performed for relations with sizes 100, 1,000 and 10,000 tuples. Input file segment sizes will be set to those values that were computed to be optimal for the projection and selection operations (256 bytes for 100 tuples relation, 2,048 bytes for the 1,000 tuple relation and 16,384 bytes for the 10,000 tuples relation). Tables 34 through 36 show the execution times for all 3 runs.

Table 34
10,000 Tuple Scalar Aggregate

No. of Processors	Time	Effective Processors	Efficiency
1	7.53	1.0	1.0000
2	3.81	1.9	0.9882
3	2.55	2.9	0.9842
4	1.96	3.8	0.9613
5	1.59	4.7	0.9458
6	1.33	5.6	0.9410
7	1.15	6.5	0.9315
8	1.02	7.3	0.9226
9	0.92	8.1	0.9069
10	0.83	8.9	0.8996
11	0.78	9.6	0.8769
12	0.70	10.6	0.8886
13	0.67	11.1	0.8597
14	0.61	12.3	0.8817
15	0.59	12.6	0.8465
16	0.58	12.8	0.8032
17	0.51	14.6	0.8614
18	0.51	14.6	0.8147
19	0.48	15.6	0.8220
20	0.45	16.5	0.8290
21	0.45	16.4	0.7815
22	0.43	17.3	0.7871
23	0.41	17.9	0.7822
24	0.40	18.6	0.7781
25	0.39	18.9	0.7569
26	0.38	19.5	0.7528
27	0.35	21.1	0.7830
28	0.34	21.5	0.7711
29	0.36	20.9	0.7207
30	0.32	23.2	0.7757

Table 35
1,000 Tuple Scalar Aggregate

No. of Processors	Time	Effective Processors	Efficiency
1	0.92	1.0	1.0000
2	0.49	1.8	0.9424
3	0.33	2.7	0.9147
4	0.26	3.4	0.8720
5	0.22	4.1	0.8346
6	0.18	5.0	0.8349
7	0.15	5.9	0.8461
8	0.16	5.7	0.7208
9	0.13	6.9	0.7775
10	0.12	7.6	0.7665
11	0.12	7.5	0.6904
12	0.11	8.3	0.6955
13	0.11	8.3	0.6399
14	0.10	8.8	0.6351
15	0.09	9.2	0.6194
16	0.08	10.3	0.6449
17	0.09	10.2	0.6028
18	0.08	10.5	0.5863
19	0.06	13.3	0.7002
20	0.08	10.7	0.5383
21	0.08	10.9	0.5229
22	0.08	10.9	0.4962
23	0.06	13.5	0.5906
24	0.06	15.1	0.6300
25	0.05	15.6	0.6245
26	0.06	15.4	0.5936
27	0.05	16.0	0.5951
28	0.06	14.5	0.5209
29	0.05	17.5	0.6038
30	0.05	17.3	0.5775

Looking at the tables shows a maximum speedup of approximately 23.3 during the 10,000 tuples run, that figure falls off sharply as the number of tuples in the relation decreases. It is down to 9.9

Table 36 100 Tuple Scalar Aggregate			
No. of Processors	Time	Effective Processors	Efficiency
1	0.25	1.0	1.0000
2	0.12	2.0	1.0055
3	0.10	2.5	0.8391
4	0.08	3.0	0.7565
5	0.07	3.4	0.6828
6	0.06	3.9	0.6557
7	0.04	5.2	0.7527
8	0.04	5.4	0.6848
9	0.04	6.0	0.6700
10	0.05	4.7	0.4715
11	0.03	6.6	0.6086
12	0.04	6.3	0.5288
13	0.03	7.2	0.5555
14	0.04	5.4	0.3903
15	0.03	7.4	0.4935
16	0.03	7.2	0.4545
17	0.02	8.7	0.5126
18	0.04	6.1	0.3394
19	0.03	7.5	0.3989
20	0.04	5.5	0.2787
21	0.03	8.0	0.3827
22	0.04	5.3	0.2422
23	0.02	9.1	0.3973
24	0.02	9.0	0.3774
25	0.04	5.9	0.2391
26	0.02	8.7	0.3351
27	0.02	9.8	0.3651
28	0.05	5.1	0.1841
29	0.02	9.9	0.3424
30	0.02	9.9	0.3310

times speedup with 30 processors for a 100 tuple scalar aggregate operation. At this point we included the event logger so that the execution of the 100 and 1,000 tuple runs could be monitored. Figs. 5 and 6 show when all of the children finish and when the operation is actually over. As is noticed, the time necessary to retrieve and print the global result is a major component of the total time for this operation thus reducing the effective speedup for the smaller runs.

8. The Natural Join Operation

The Natural Join operation is very similar to a θ-join with θ set to "=", otherwise known as an *equijoin*. The only difference is the fact that all attributes that name an equivalent field in both input relations must be equal. After this only one occurrence of each duplicated attribute will be included in

Figure 5: Scalar aggregate for 100 tuple relation.

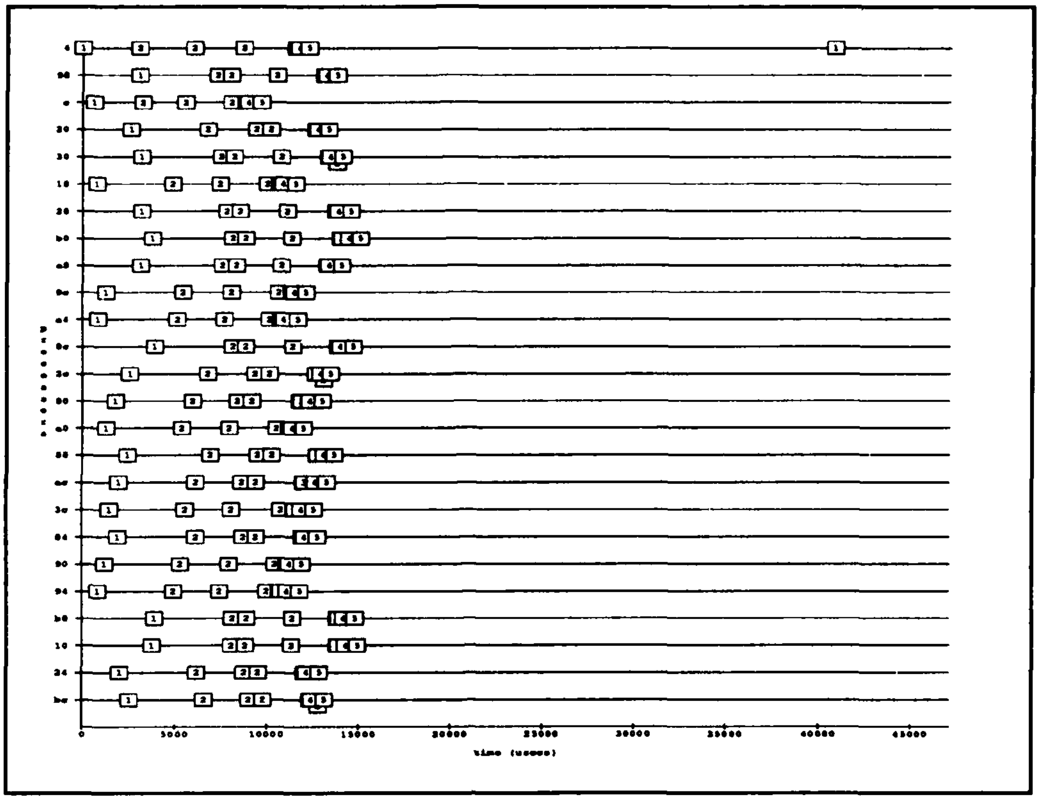
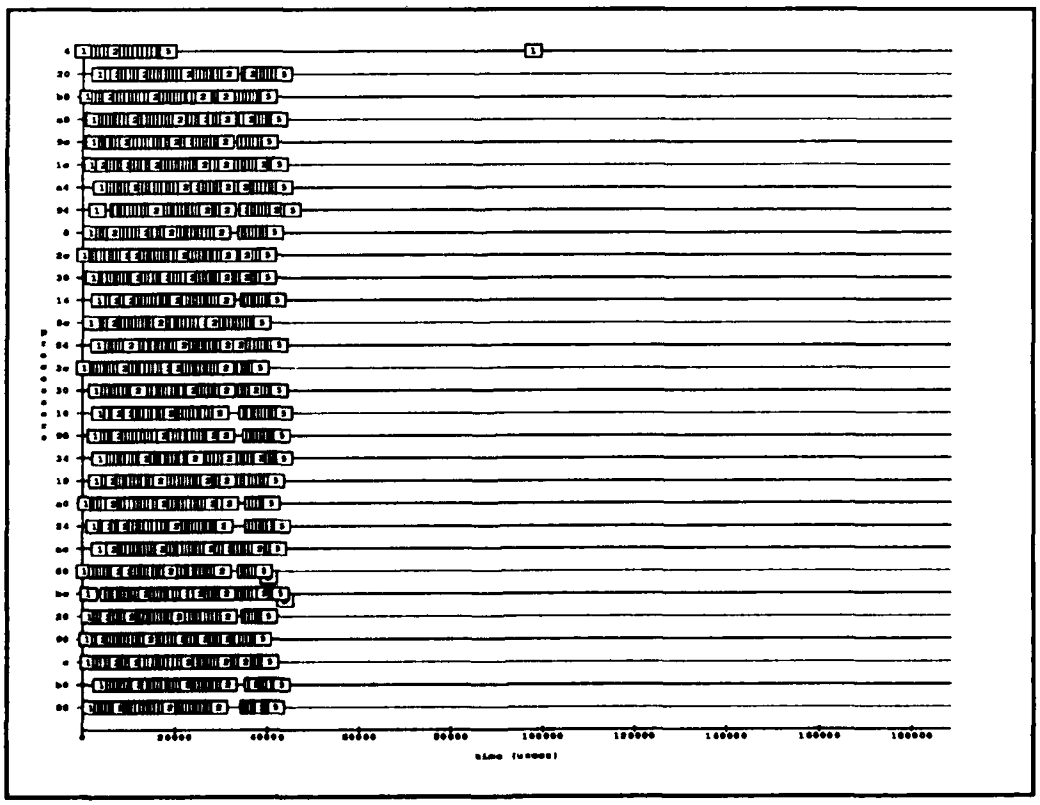


Figure 6: Scalar aggregate for 1,000 tuple relation.



the output relation. Since the natural join is very similar to the selection operation, the chained-linked hashing method was extended and incorporated into this algorithm. The natural join algorithm will consist of five different sections. These sections will be, hashing the first (smaller) relation on the duplicated fields. Hashing the second (larger) relation on the duplicated fields and comparing each hashed tuple from the second relation against all of those in the chain-link corresponding to that particular hash table entry. In the third step we will release all the memory that was occupied by the hash table for the first relation. The fourth step will consist of counting the output tuples and the allocation of the exact amount of memory for the output file. Lastly, we will unload the output hash table into the newly created output file.

8.1. Hashing the First (Smaller) Relation

This first section will consist of distributing the smaller relation throughout a chain-linked hash table by hashing each tuple on the attributes that are common. Initially a test will have to be performed on the two relations to determine which one is smaller. There are several criteria that need to be taken into consideration when deciding which file is the smallest and should be hashed first. Since the file will be hashed and stored in main memory, the actual size of the file, $\tau \times \theta$, is an important factor that must taken into consideration. Another crucial factor is the actual number of tuples in each relation. This is because all of the tuples in the second relation must be compared against all of the tuples in the first hash table. The fewer tuples there are in the first hash table will reduce the average number of tuples in each linked list.

The first consideration should be actual relation size, $\tau \times \theta$ bytes. This is because main memory will have to be dynamically allocated to hold all of the hashed tuples while each tuple in the second relation is hashed to the first hash table. When relations get very large (i.e. > 10,000 tuples and/or tuple length gets very large), the cumulative amount of memory needed can demand a lot from a system. Our Butterfly's current configuration contains 128 Mbytes of main memory, which was more than enough to hold several 10,000 tuple relations (in RAMFile form) and the two hash tables each containing all of the tuples being joined in each.

The second consideration is the actual number of tuples in each relation. This is because the time needed to allocate and process each tuple into the first hash table will be a major portion of the the execution time for this algorithm. It will be much faster to allocate and hash 1,000 tuples with size 500 bytes than it will to allocate and hash 5,000 tuples with size 100 bytes. Another reason for hashing the relation with the smaller number of tuples is that the fewer the number of tuples to be hashed the shorter the average length of the linked list at each hash index. This second fact will play an important role in the second step when each tuple in the second relation will have to be compared against every tuple in the linked list to which it has been hashed.

8.2. Hashing and Matching Second Relation

Hashing and matching the second relation with the first relation will be the most complicated step in the natural join algorithm. During this step we will divide the second relation among the ρ processors, with each processor getting δ tuples (using the same δ as above). For every tuple in its block, a task will compute the hash value of the attributes in the second relation's tuple that both relations have in common so that tuples from both relations that match all common attributes will hash to the same index in the first hash table. This will give us the index to the only chain of tuples from the first relation that the tuple from the second relation can possibly be joined with. Hashing the relation with the lesser amount of tuples first will keep the average length of this chain to a minimum thereby minimizing the number of tuples that must be compared against the tuple from the second relation. The complete chain of tuples must be tested for a possible match in case there are multiple matches. For every match that does occur, a new tuple large enough to contain all of the attributes from both relations must be allocated from dynamic memory. Next, all necessary data fields must be copied into the output tuple. Lastly, this tuple must be stored somewhere until the actual number of joined tuples can be accounted for and a RAMFile can be allocated in main memory.

We decided to rehash this new output tuple on all the attributes and with a different hash table size. This was done to try to redistribute the new tuples throughout memory so that they will be easily divided among all of the processors when it becomes necessary to output the joined tuples into

the newly created output relation. The second hash table has a different size than the first table to help redistribute the tuples throughout the table. After the hash index for the output tuple has been computed into the second hash table, the operation of inserting the new tuple into the linked list at that particular index must be performed atomically to prevent any consistency problems from occurring and then a counter field must be incremented.

8.3. Releasing Tuples From the First Hash Table

Now that the two relations have been joined and their combined tuples have been stored in a different hash table, the first hash table and all of its linked lists will not be needed any more. The amount of memory used to store the first relation in the first hash table may be quite large and it would be very wasteful (if not perilous to the system) to just abandon it. Using the same δ as above, we will divide all of the hash table's linked lists between all of the processors and then have each task traverse each linked list releasing each tuple in turn. This step will be combined with the following and will be discussed below.

8.4. Counting Output Tuples and Allocating Correct Memory

As mentioned above, each index in the second hash table also contains a counter field that will maintain a count of the number of tuples that are in that particular linked list. From this field we will be able to count the number of tuples that have been joined and must have space allocated for them. The operation of how we went about counting the tuples and creating the new RAMFile has been discussed in the previous algorithms. A common serial problem in many of the above algorithms has been when the algorithms have been performed for the smaller sized relations, the time necessary to create a new RAMFile does not decrease when more processors are added and becomes a major contributor to the execution time for the query.

8.5. Unloading the Second Hash Table into the Output File

In this final step we will unload the second hash table into the newly created output file in parallel. Each task will be given a block of entries from the second hash table from which to unload tuples. The number of entries, δ, will be determined using the formula given above, but instead of dividing

tuples we will be dividing hash table entries. For each entry that a task has to operate on, it will traverse the linked list outputting each tuple and releasing the memory that was associated with it.

To determine where each task will be placing its block of tuples, a task will first sum up all of the count fields within its block of entries. This will give it the number of tuples for which to reserve space. Then it will reserve space by atomically retrieving and incrementing a global EOF pointer. It will increment the EOF pointer by the number of tuples in its block causing the next task to start at the position after the previous task's last tuple.

8.6. Natural Join Results

The natural join algorithm has been run with the following sets of input and output relation sizes in number of tuples: 10,000 $\times$ 10,000 giving 10,000, 10,000 $\times$ 1,000 giving 1,000, 1,000 $\times$ 1,000 giving 1,000, 1,000 $\times$ 100 giving 100 and 100 $\times$ 100 giving 100. These results can be seen in Table 37 through 41.

The results for the natural join again suffer from the serial RAMFile creation time. This was partially alleviated by combining the RAMFile creation with the releasing tuples from the first hash table stage. Both of these stages are independent of each other and may be performed concurrently. Since the RAMFile creation must be performed by only one processor all other remaining idle processors can be put to work releasing tuples from the hash table. In the case when only one processor is being utilized these two operations must proceed one at a time. This will place a lower limit on the time it takes to perform both operations of the time it takes to create the RAMFile. By looking at the intermediate times, we see that this increases the efficiency of the algorithm and significantly reduces the execution times of the larger test cases. It does not greatly improve the times for the smaller joined relations because the time necessary to create the new RAMFile is still significantly greater than the time saved by placing the freeing up memory routine in the background running in parallel.

9. Future Work

In this paper we discussed the implementation of basic database operations on the Butterfly parallel processor. Although there has been some consideration given to this problem for cube-

Table 37 10,000 × 10,000 Tuple Natural Join Resulting Relation Contains: 10,000 Tuples			
No. of Processors	Time	Effective Processors	Efficiency
1	118.89	1.0	1.0000
2	60.48	1.9	0.9828
3	41.50	2.8	0.9547
4	31.80	3.7	0.9344
5	24.73	4.8	0.9613
6	21.55	5.5	0.9193
7	17.54	6.7	0.9683
8	17.16	6.9	0.8655
9	15.72	7.5	0.8402
10	13.93	8.5	0.8533
11	12.48	9.5	0.8658
12	11.40	10.4	0.8688
13	10.49	11.3	0.8715
14	9.68	12.2	0.8765
15	9.04	13.1	0.8766
16	8.33	14.2	0.8911
17	8.01	14.8	0.8729
18	7.51	15.8	0.8792
19	7.19	16.5	0.8695
20	6.88	17.2	0.8639
21	6.63	17.9	0.8536
22	6.34	18.7	0.8517
23	6.21	19.1	0.8312
24	5.94	19.9	0.8328
25	5.83	20.3	0.8156
26	5.64	21.0	0.8103
27	5.46	21.7	0.8051
28	5.40	22.0	0.7858
29	5.28	22.5	0.7760
30	5.17	22.9	0.7662

Table 38 10,000 × 1,000 Tuple Natural Join Resulting Relation Contains: 1,000 Tuples			
No. of Processors	Time	Effective Processors	Efficiency
1	24.03	1.0	1.0000
2	11.85	2.0	1.0135
3	7.49	3.2	1.0688
4	5.87	4.0	1.0225
5	4.89	4.9	0.9819
6	4.28	5.6	0.9349
7	3.82	6.2	0.8984
8	3.47	6.9	0.8637
9	3.23	7.4	0.8262
10	3.02	7.9	0.7941
11	2.84	8.4	0.7676
12	2.73	8.7	0.7331
13	2.60	9.2	0.7087
14	2.53	9.4	0.6775
15	2.43	9.8	0.6592
16	2.36	10.1	0.6345
17	2.29	10.4	0.6169
18	2.25	10.6	0.5913
19	2.18	10.9	0.5781
20	2.16	11.0	0.5538
21	2.13	11.2	0.5358
22	2.11	11.3	0.5159
23	2.07	11.5	0.5029
24	2.08	11.5	0.4794
25	2.02	11.8	0.4740
26	2.03	11.8	0.4546
27	2.02	11.8	0.4403
28	2.01	11.9	0.4257
29	1.96	12.2	0.4221
30	1.98	12.1	0.4037

connected computers [2,7], this work to our knowledge represents the first attempt to implement these operations on an existing general-purpose parallel machine. We are continuing to examine the parallel algorithms used to implement the database operations, and it is likely that a more efficient implementation will emerge in the future.

It is also important to realize that the results of our experiments are based on version 3.0 of the Chrysalis operating system which provides support for applications programs on the Butterfly. BBN expects to be coming out with a new version of Chrysalis in the near future which may affect the implementation decisions that were made in this paper. However, it is our belief that much of our analysis in this paper dealing with the problems of improper data division and hot spots will remain

Table 39 1,000 × 1,000 Tuple Natural Join Resulting Relation Contains: 1,000 Tuples			
No. of Processors	Time	Effective Processors	Efficiency
1	7.33	1.0	1.0000
2	4.47	1.6	0.8201
3	3.28	2.2	0.7450
4	2.60	2.8	0.7041
5	2.25	3.2	0.6500
6	2.03	3.6	0.6016
7	1.87	3.9	0.5574
8	1.75	4.1	0.5223
9	1.69	4.3	0.4812
10	1.64	4.4	0.4459
11	1.57	4.6	0.4243
12	1.55	4.7	0.3936
13	1.50	4.8	0.3738
14	1.50	4.8	0.3486
15	1.46	4.9	0.3329
16	1.42	5.1	0.3219
17	1.41	5.1	0.3053
18	1.44	5.0	0.2817
19	1.43	5.1	0.2690
20	1.37	5.3	0.2665
21	1.34	5.4	0.2599
22	1.37	5.3	0.2424
23	1.37	5.3	0.2318
24	1.33	5.4	0.2287
25	1.42	5.1	0.2057
26	1.36	5.3	0.2061
27	1.43	5.1	0.1891
28	1.37	5.3	0.1899
29	1.35	5.4	0.1864
30	1.37	5.3	0.1772

Table 40 1,000 × 100 Tuple Natural Join Resulting Relation Contains: 100 Tuples			
No. of Processors	Time	Effective Processors	Efficiency
1	3.16	1.0	1.0000
2	2.07	1.5	0.7626
3	1.76	1.7	0.5985
4	1.54	2.0	0.5102
5	1.47	2.1	0.4281
6	1.44	2.1	0.3642
7	1.40	2.2	0.3208
8	1.36	2.3	0.2889
9	1.38	2.2	0.2546
10	1.33	2.3	0.2364
11	1.35	2.3	0.2124
12	1.40	2.2	0.1873
13	1.41	2.2	0.1725
14	1.35	2.3	0.1670
15	1.35	2.3	0.1550
16	1.38	2.2	0.1431
17	1.36	2.3	0.1365
18	1.40	2.2	0.1252
19	1.45	2.1	0.1146
20	1.41	2.2	0.1119
21	1.41	2.2	0.1067
22	1.40	2.2	0.1022
23	1.49	2.1	0.0918
24	1.48	2.1	0.0888
25	1.49	2.1	0.0844
26	1.49	2.1	0.0812
27	1.47	2.1	0.0794
28	1.52	2.0	0.0742
29	1.53	2.0	0.0709
30	1.54	2.0	0.0682

valid even with the new Chrysalis release.

References

[1] Selim G. Akl, *Parallel Sorting Algorithms*, Academic Press, Orlando, 1985.

[2] Chaitanya K. Baru and Ophir Frieder, "Implementing relational database operations in a cube-connected multicomputer," *Proc. 3rd IEEE Int'l. Conf. on Data Engineering*, pp. 36-43, February 1987.

[3] BBN Advanced Computers Inc., "Butterfly Parallel Processor - Overview," BBN Report No. 6148, BBN Laboratories Incorporated, Mar. 6, 1986.

Table 41 100 × 100 Tuple Natural Join Resulting Relation Contains: 100 Tuples			
No. of Processors	Time	Effective Processors	Efficiency
1	1.85	1.0	1.0000
2	1.45	1.2	0.6378
3	1.28	1.4	0.4814
4	1.25	1.4	0.3710
5	1.18	1.5	0.3147
6	1.15	1.6	0.2678
7	1.20	1.5	0.2199
8	1.22	1.5	0.1898
9	1.30	1.4	0.1581
10	1.28	1.4	0.1444
11	1.30	1.4	0.1295
12	1.32	1.4	0.1167
13	1.34	1.3	0.1064
14	1.35	1.3	0.0982
15	1.33	1.3	0.0926
16	1.33	1.3	0.0868
17	1.30	1.4	0.0839
18	1.35	1.3	0.0760
19	1.35	1.3	0.0719
20	1.34	1.3	0.0692
21	1.35	1.3	0.0650
22	1.36	1.3	0.0620
23	1.45	1.2	0.0553
24	1.46	1.2	0.0528
25	1.40	1.3	0.0527
26	1.40	1.3	0.0509
27	1.44	1.2	0.0475
28	1.48	1.2	0.0447
29	1.45	1.2	0.0440
30	1.44	1.2	0.0427

[4] BBN Advanced Computers Inc., "Butterfly Parallel Processor - The Uniform System Approach to Programming the Butterfly Parallel Processor," BBN Report No. 6149 version 2, BBN Laboratories Incorporated, Oct. 10, 1986.

[5] BBN Advanced Computers Inc., "Butterfly Parallel Processor - The Butterfly RAMFile System," BBN Report No. 6351, BBN Laboratories Incorporated, Sept. 10, 1986.

[6] Dina Bitton, David J. DeWitt and Carolyn Turbyfill, "Benchmarking database systems: A systematic approach," Technical Report #526. Computer Science Department, University of Wisconsin, December 1983.

[7] Ophir Frieder and Chaitanya K. Baru, "Data distribution and query scheduling policies for a cube-connected multicomputer system," The University of Michigan Computing Research Laboratory.

[8] W. Daniel Hillis, *The Connection Machine,* The MIT Press, Cambridge, 1985.

[9] Randall Rettberg and Robert Thomas, "Contention is no obstacle to shared-memory multiprocessing," *Comm. of the ACM,* vol. 29, no. 12, pp. 1202-1212, December 1986.

[10] David L. Waltz, "Applications of the Connection Machine," *IEEE Computer,* vol. 20, no. 1, pp. 85-97, January 1987.

END
DATE
FILMED
DTIC
6-88